Bareos Developer Guide Documentation

Release 17.2.7

Bareos GmbH & Co. KG

Jan 10, 2019

CONTENTS:

CHAPTER

ONE

BAREOS DEVELOPER NOTES

This document is intended mostly for developers and describes how you can contribute to the Bareos project and the general framework of making Bareos source changes.

Fig. 1.1: This is an example UML diagram

1.1 History

Bareos is a fork of the open source project Bacula version 5.2. In 2010 the Bacula community developer Marco van Wieringen started to collect rejected or neglected community contributions in his own branch. This branch was later on the base of Bareos and since then was enriched by a lot of new features.

This documentation also bases on the original Bacula documentation, it is technically also a fork of the documenation created following the rules of the GNU Free Documentation License.

Original author of Bacula and it's documentation is Kern Sibbald. We thank Kern and all contributors to Bacula and it's documentation. We maintain a list of contributors to Bacula (until the time we've started the fork) and Bareos in our AUTHORS file.

1.2 Contributions

Contributions to the Bareos project come in many forms: ideas, participation in helping people on the bareos-users email list, packaging Bareos binaries for the community, helping improve the documentation, and submitting code.

1.3 Patches

Subject to the copyright assignment described below, your patches should be sent in **git format-patch** format relative to the current contents of the master branch of the Git repository. Please attach the output file or files generated by the **git format-patch** to the email rather than include them direct to avoid wrapping of the lines in the patch. Please be sure to use the Bareos indenting standard (see below) for source code. If you have checked out the source with Git, you can get a diff using.

```
git pull
git format-patch -M
```

If you plan on doing significant development work over a period of time, after having your first patch reviewed and approved, you will be eligible for having developer Git write access so that you can commit your changes directly to the Git repository. To do so, you will need a userid on Source Forge.

1.3.1 Bugs Database

We have a bugs database which is at https://bugs.bareos.org.

The first thing is if you want to take over a bug, rather than just make a note, you should assign the bug to yourself. This helps other developers know that you are the principal person to deal with the bug. You can do so by going into the bug and clicking on the **Update Issue** button. Then you simply go to the **Assigned To** box and select your name from the drop down box. To actually update it you must click on the **Update Information** button a bit further down on the screen, but if you have other things to do such as add a Note, you might wait before clicking on the **Update Information** button.

Generally, we set the **Status** field to either acknowledged, confirmed, or feedback when we first start working on the bug. Feedback is set when we expect that the user should give us more information.

Normally, once you are reasonably sure that the bug is fixed, and a patch is made and attached to the bug report, and/or in the SVN, you can close the bug. If you want the user to test the patch, then leave the bug open, otherwise close it and set **Resolution** to **Fixed**. We generally close bug reports rather quickly, even without confirmation, especially if we have run tests and can see that for us the problem is fixed. However, in doing so, it avoids misunderstandings if you leave a note while you are closing the bug that says something to the following effect: We are closing this bug because ... If for some reason, it does not fix your problem, please feel free to reopen it, or to open a new bug report describing the problem".

We do not recommend that you attempt to edit any of the bug notes that have been submitted, nor to delete them or make them private. In fact, if someone accidentally makes a bug note private, you should ask the reason and if at all possible (with his agreement) make the bug note public.

If the user has not properly filled in most of the important fields (platorm, OS, Product Version, ...) please do not hesitate to politely ask him. Also, if the bug report is a request for a new feature, please politely send the user to the Feature Request menu item on www.bareos.org. The same applies to a support request (we answer only bugs), you might give the user a tip, but please politely refer him to the manual and the Getting Support page of www.bareos.org.

1.3.2 Developing Bareos

1.4 Compiling

There are several ways to locally compile (and install) Bareos

```
sudo apt-get install git dpkg-dev devscripts fakeroot
git clone https://github.com/bareos/bareos
cd bareos/core
dpkg-checkbuilddeps
```

You then need to install all packages that dpkg-checkbuilddeps lists as required

```
# prepares the changelog for Debian, only neccesary on initial install
cp -a platforms/packaging/bareos.changes debian/changelog
# You need to manually change the version number in debian/changelog.
# gets current version number from src/include/version.h and includes it
VERSION=$(sed -n -r 's/#define VERSION "(.*)"/\1/p'  src/include/version.h)
dch -v $VERSION "Switch version number"
```

```
# creates Debian-packages and stores them in ..
fakeroot debian/rules binary
```

Disclaimer: This process makes use of development-oriented compiler flags. If you want to compile Bareos to be similar to a Bareos compiled with production intent, please refer to section "Using the same flags as in production".

```
git clone https://github.com/bareos/bareos
cd bareos/core
```

```
#!/bin/bash
export CFLAGS="-g -Wall"
export CXXFLAGS="-g -Wall"

# specifies the directory in which bareos will be installed
DESTDIR=~/bareos

mkdir $DESTDIR

CMAKE_BUILDDIR=cmake-build

mkdir ${CMAKE_BUILDDIR}
pushd ${CMAKE_BUILDDIR}

# In a normal installation, Dbaseport=9101 is used. However, for testing purposes, we␣
↪make use of port 8001.
cmake  .. \
  -DCMAKE_VERBOSE_MAKEFILE=ON \
  -DBUILD_SHARED_LIBS:BOOL=ON \
  -Dbaseport=8001 \
  -DCMAKE_INSTALL_PREFIX:PATH=$DESTDIR \
  -Dprefix=$DESTDIR \
  -Dworkingdir=$DESTDIR/var/ \
  -Dpiddir=$DESTDIR/var/ \
  -Dconfigtemplatedir=$DESTDIR/lib/defaultconfigs \
  -Dsbin-perm=755 \
  -Dpython=yes \
  -Dsmartalloc=yes \
  -Ddisable-conio=yes \
  -Dreadline=yes \
  -Dbatch-insert=yes \
  -Ddynamic-cats-backends=yes \
  -Ddynamic-storage-backends=yes \
  -Dscsi-crypto=yes \
  -Dlmdb=yes \
  -Dndmp=yes \
  -Dipv6=yes \
  -Dacl=yes \
  -Dxattr=yes \
  -Dpostgresql=yes \
  -Dmysql=yes \
  -Dsqlite3=yes \
  -Dtcp-wrappers=yes \
  -Dopenssl=yes \
  -Dincludes=yes

popd
```

You will now have to do the following:

```
# This path corresponds to the $CMAKE_BUILDDIR variable. If you used a directory␣
↪other than the default ```cmake-build```, you will have to alter the path␣
↪accordingly.
cd cmake-build
make
make install
```

Before you can successfully use your local installation, it requires additional configuration.

```
# You have to move to the local installation directory. This path corresponds to the
↪$DESTDIR variable. If you used a directory other than the default ```~/baroes```,␣
↪you will have to alter the path accordingly.
cd ~/bareos
# copy configuration files, only neccesary on initial install
cp -a lib/defaultconfigs/* etc/bareos/
```

You will have to replace `dbdriver = "XXX_REPLACE_WITH_DATABASE_DRIVER_XXX"` with `sqlite3` or other. The file can be found at `etc/bareos/bareos-dir.d/catalog/MyCatalog.conf`

```
# sets up server
# creates bareos database (requires sqlite3 package in case of sqlite3 installation)
lib/bareos/scripts/create_bareos_database
lib/bareos/scripts/make_bareos_tables
lib/bareos/scripts/grant_bareos_privileges
```

```
# launches director in debug mode in foreground
sbin/bareos-dir -f -d100
# displays status of bareos daemons
lib/bareos/scripts/bareos status
# The start command launches both the daemons and the director, if not already␣
↪launched. We launched the director seperately for debugging purposes.
lib/bareos/scripts/bareos start '
# launches bconsole to connect to director
bin/bconsole
```

You can find the compilation flags that are used in production in the following locations:

You can find the flags used for compiling for Debian in debian/rules.

You can find the flags used for compiling rpm-packages in core/platforms/packaging/bareos.spec.

1.5 Debugging

Probably the first thing to do is to turn on debug output.

A good place to start is with a debug level of 20 as in **./startit -d20**. The startit command starts all the daemons with the same debug level. Alternatively, you can start the appropriate daemon with the debug level you want. If you really need more info, a debug level of 60 is not bad, and for just about everything a level of 200.

1.6 Using a Debugger

If you have a serious problem such as a segmentation fault, it can usually be found quickly using a good multiple thread debugger such as **gdb**. For example, suppose you get a segmentation violation in **bareos-dir**. You might use

the following to find the problem:

<start the Storage and File daemons> cd dird gdb ./bareos-dir run -f -s -c ./dird.conf <it dies with a segmentation fault> where The **-f** option is specified on the **run** command to inhibit **dird** from going into the background. You may also want to add the **-s** option to the run command to disable signals which can potentially interfere with the debugging.

As an alternative to using the debugger, each **Bareos** daemon has a built in back trace feature when a serious error is encountered. It calls the debugger on itself, produces a back trace, and emails the report to the developer. For more details on this, please see the chapter in the main Bareos manual entitled "What To Do When Bareos Crashes (Kaboom)".

1.7 Memory Leaks

Because Bareos runs routinely and unattended on client and server machines, it may run for a long time. As a consequence, from the very beginning, Bareos uses SmartAlloc to ensure that there are no memory leaks. To make detection of memory leaks effective, all Bareos code that dynamically allocates memory MUST have a way to release it. In general when the memory is no longer needed, it should be immediately released, but in some cases, the memory will be held during the entire time that Bareos is executing. In that case, there MUST be a routine that can be called at termination time that releases the memory. In this way, we will be able to detect memory leaks. Be sure to immediately correct any and all memory leaks that are printed at the termination of the daemons.

1.8 When Implementing Incomplete Code

Please identify all incomplete code with a comment that contains

```
***FIXME***
```

where there are three asterisks (*) before and after the word FIXME (in capitals) and no intervening spaces. This is important as it allows new programmers to easily recognize where things are partially implemented.

1.9 Header Files

Please carefully follow the scheme defined below as it permits in general only two header file includes per C file, and thus vastly simplifies programming. With a large complex project like Bareos, it isn't always easy to ensure that the right headers are invoked in the right order (there are a few kludges to make this happen – i.e. in a few include files because of the chicken and egg problem, certain references to typedefs had to be replaced with **void**).

Every file should include **bareos.h**. It pulls in just about everything, with very few exceptions. If you have system dependent ifdefing, please do it in **baconfig.h**. The version number and date are kept in **version.h**.

Each of the subdirectories (console, cats, dird, filed, findlib, lib, stored, ...) contains a single directory dependent include file generally the name of the directory, which should be included just after the include of **bareos.h**. This file (for example, for the dird directory, it is **dird.h**) contains either definitions of things generally needed in this directory, or it includes the appropriate header files. It always includes **protos.h**. See below.

Each subdirectory contains a header file named **protos.h**, which contains the prototypes for subroutines exported by files in that directory. **protos.h** is always included by the main directory dependent include file.

1.10 Programming Standards

For the most part, all code should be written in C unless there is a burning reason to use C++, and then only the simplest C++ constructs will be used. Note, Bareos is slowly evolving to use more and more C++.

Code should have some documentation – not a lot, but enough so that I can understand it. Look at the current code, and you will see that I document more than most, but am definitely not a fanatic.

We prefer simple linear code where possible. Gotos are strongly discouraged except for handling an error to either bail out or to retry some code, and such use of gotos can vastly simplify the program.

Remember this is a C program that is migrating to a **tiny** subset of C++, so be conservative in your use of C++ features.

1.11 Do Not Use

- STL – it is totally incomprehensible.

1.12 Avoid if Possible

- Using **void *** because this generally means that one must using casting, and in C++ casting is rather ugly. It is OK to use void * to pass structure address where the structure is not known to the routines accepting the packet (typically callback routines). However, declaring "void *buf" is a bad idea. Please use the correct types whenever possible.
- Using undefined storage specifications such as (short, int, long, long long, size_t ...). The problem with all these is that the number of bytes they allocate depends on the compiler and the system. Instead use Bareos's types (int8_t, uint8_t, int32_t, uint32_t, int64_t, and uint64_t). This guarantees that the variables are given exactly the size you want. Please try at all possible to avoid using size_t ssize_t and the such. They are very system dependent. However, some system routines may need them, so their use is often unavoidable.
- Returning a malloc'ed buffer from a subroutine – someone will forget to release it.
- Heap allocation (malloc) unless needed – it is expensive. Use POOL_MEM instead.
- Templates – they can create portability problems.
- Fancy or tricky C or C++ code, unless you give a good explanation of why you used it.
- Too much inheritance – it can complicate the code, and make reading it difficult (unless you are in love with colons)

1.13 Do Use Whenever Possible

- Locking and unlocking within a single subroutine.
- A single point of exit from all subroutines. A goto is perfectly OK to use to get out early, but only to a label named bail_out, and possibly an ok_out. See current code examples.
- malloc and free within a single subroutine.
- Comments and global explanations on what your code or algorithm does.
- When committing a fix for a bug, make the comment of the following form:

```
Fixes #1234: Description of the bug.

Reason for bug fix
or other message.
```

It is important to write the **bug #1234** like that because our program that automatically pulls messages from the git repository to make the ChangeLog looks for that pattern. Obviously the **1234** should be replaced with the number of the bug you actually fixed.

Providing the commit comment line has one of the following keywords (or phrases), it will be ignored:

```
tweak
typo
cleanup
bweb:
regress:
again
.gitignore
fix compilation
technotes
update version
update technotes
update projects
update releasenotes
update version
update home
update release
update todo
update notes
update changelog
```

- Use the following keywords at the beginning of a git commit message

1.14 Indenting Standards

We find it very hard to read code indented 8 columns at a time. Even 4 at a time uses a lot of space, so we have adopted indenting 3 spaces at every level. Note, indention is the visual appearance of the source on the page, while tabbing is replacing a series of up to 8 spaces from a tab character.

The closest set of parameters for the Linux **indent** program that will produce reasonably indented code are:

```
indent -nbad -bap -bbo -nbc -br -brs -c36 -cd36 -ncdb -ce -ci3 -cli0 -cp36 -d0 -di1 -
↪ndj -nfc1 -nfca -hnl -i3 -ip0 -l85 -lp -npcs -nprs -npsl -saf -sai -saw -nsob -nss -
↪nbc -ncs -nbfda --no-tabs -T POOL_MEM -T json_t
```

You can put the above in your .indent.pro file, and then just invoke indent on your file. However, be warned. This does not produce perfect indenting, and it will mess up C++ class statements pretty badly.

Also typedefs/classes must be specified by the `-T` flag.

Braces are required in all if statements (missing in some very old code). To avoid generating too many lines, the first brace appears on the first line (e.g. of an if), and the closing brace is on a line by itself. E.g.

```
 if (abc) {
    some_code;
}
```

Just follow the convention in the code. For example we I prefer non-indented cases.

```
 switch (code) {
 case 'A':
    do something
    break;
 case 'B':
    again();
    break;
 default:
    break;
}
```

Avoid using // style comments except for temporary code or turning off debug code. Standard C comments are preferred (this also keeps the code closer to C).

Attempt to keep all lines less than 85 characters long so that the whole line of code is readable at one time. This is not a rigid requirement.

Always put a brief description at the top of any new file created describing what it does and including your name and the date it was first written. Please don't forget any Copyrights and acknowledgments if it isn't 100% your code. Also, include the Bareos copyright notice that is in **src/c**.

In general you should have two includes at the top of the an include for the particular directory the code is in, for includes are needed, but this should be rare.

In general (except for self-contained packages), prototypes should all be put in **protos.h** in each directory.

Always put space around assignment and comparison operators.

```
 a = 1;
 if (b >= 2) {
   cleanup();
}
```

but your can compress things in a **for** statement:

```
for (i=0; i < del.num_ids; i++) {
 ...
```

Don't overuse the inline if (?:). A full **if** is preferred, except in a print statement, e.g.:

```
 if (ua->verbose \&& del.num_del != 0) {
    bsendmsg(ua, _("Pruned %d %s on Volume %s from catalog.\n"), del.num_del,
       del.num_del == 1 ? "Job" : "Jobs", mr->VolumeName);
}
```

Leave a certain amount of debug code (Dmsg) in code you submit, so that future problems can be identified. This is particularly true for complicated code likely to break. However, try to keep the debug code to a minimum to avoid bloating the program and above all to keep the code readable.

Please keep the same style in all new code you develop. If you include code previously written, you have the option of leaving it with the old indenting or re-indenting it. If the old code is indented with 8 spaces, then please re-indent it to Bareos standards.

If you are using **vim**, simply set your tabstop to 8 and your shiftwidth to 3.

1.15 Tabbing

Tabbing (inserting the tab character in place of spaces) is as normal on all Unix systems – a tab is converted space up to the next column multiple of 8. My editor converts strings of spaces to tabs automatically – this results in significant compression of the files. Thus, you can remove tabs by replacing them with spaces if you wish. Please don't confuse tabbing (use of tab characters) with indenting (visual alignment of the code).

1.16 Don'ts

Please don't use:

```
strcpy()
strcat()
strncpy()
strncat();
sprintf()
snprintf()
```

They are system dependent and un-safe. These should be replaced by the Bareos safe equivalents:

```
char *bstrncpy(char *dest, char *source, int dest_size);
char *bstrncat(char *dest, char *source, int dest_size);
int bsnprintf(char *buf, int32_t buf_len, const char *fmt, ...);
int bvsnprintf(char *str, int32_t size, const char  *format, va_list ap);
```

See src/lib/bsys.c for more details on these routines.

Don't use the **%lld** or the **%q** printf format editing types to edit 64 bit integers – they are not portable. Instead, use **%s** with **edit_uint64()**. For example:

```
char buf[100];
uint64_t num = something;
char ed1[50];
bsnprintf(buf, sizeof(buf), "Num=%s\n", edit_uint64(num, ed1));
```

Note: **%lld** is now permitted in Bareos code – we have our own printf routines which handle it correctly. The edit_uint64() subroutine can still be used if you wish, but over time, most of that old style will be removed.

The edit buffer **ed1** must be at least 27 bytes long to avoid overflow. See src/lib/edit.c for more details. If you look at the code, don't start screaming that I use **lld**. I actually use subtle trick taught to me by John Walker. The **lld** that appears in the editing routine is actually **#define** to a what is needed on your OS (usually "lld" or "q") and is defined in autoconf/configure.in for each OS. C string concatenation causes the appropriate string to be concatenated to the "%".

Also please don't use the STL or Templates or any complicated C++ code.

CHAPTER

TWO

MESSAGE CLASSES

Currently, there are following classes of messages:

- Memory Messages
- Debug Messages
- Job Messages
- Queued Job Messages
- Error Messages

2.1 Memory Messages

Memory messages are messages that are edited into a memory buffer. Generally they are used in low level routines. These routines do not generally have access to the Job Control Record and so they return messages reformatted in a memory buffer.

```
Mmsg(resultmessage, "3603 JobId=%u device %s is busy reading.\n", jobid);
```

2.2 Debug Messages

Debug messages are designed to be turned on at a specified debug level and are always sent to STDOUT. There are designed to only be used in the development debug process. They are coded as:

```
DmsgN(level, message, arg1, ...)
```

where

- `N` is a number indicating how many arguments are to be substituted into the message (i.e. it is a count of the number arguments you have in your message – generally the number of percent signs (%)).
- `level` is the debug level at which you wish the message to be printed.
- `message` is the message to be printed, and
- `arg1,...` are the arguments to be substituted.

Since not all compilers support #defines with varargs, you must explicitly specify how many arguments you have.

When the debug message is printed, it will automatically be prefixed by the name of the daemon which is running, the filename where the Dmsg is, and the line number within the file.

Some actual examples are:

```
Dmsg2(20, "MD5len=%d MD5=%s\n", strlen(buf), buf);
Dmsg1(9, "Created client %s record\n", client->hdr.name);
```

2.3 Job Messages

Job messages are messages that pertain to a particular job such as a file that could not be saved, or the number of files and bytes that were saved. They are coded as:

```
Jmsg(JCR, ERROR_CODE, 0, message, arg1, ...);
```

where

- `JCR` is the **Job Control Record**. If `JCR == NULL` and the `JCR` can not be determined, the message is treated as Daemon message, with fallback to output to `STDOUT`.
- `ERROR_CODE` indicates the severity or class of error
- `0` might be a timestamp, but is generally 0 (timestamp will be set to current time)
- `message` is the message to be printed, and
- `arg1,...` are the arguments to be substituted.

`ERROR_CODE` is one of the following:

ERROR OR CODE	Description
M_ABORT	Causes the daemon to immediately abort. This should be used only in extrem e cases. It attempts to produce a traceback.
M_ERROR_TERM	Causes the daemon to immediately terminate. This should be used only in extreme cases. It does not produce a traceback.
M_FATAL	Causes the daemon to terminate the current job, but the daemon keeps running.
M_ERROR	Reports the error. The daemon and the job continue running.
M_WARNING	Reports an warning message. The daemon and the job continue running.
M_INFO	Reports an informational message.

The Jmsg() takes varargs so can have any number of arguments for substituted in a printf like format. Output from the Jmsg() will go to the Job report.

If the Jmsg is followed with a number such as Jmsg1(...), the number indicates the number of arguments to be substituted (varargs is not standard for `#defines`), and what is more important is that the file and line number will be prefixed to the message. This permits a sort of debug from user's output.

```
Jmsg1(jcr, M_WARNING, 0, "Plugin=%s not found.\n", cmd);
Jmsg1(jcr, M_ERROR, 0, "%s exists but is not a directory.\n", path);
Jmsg0(NULL, M_ERROR_TERM, 0, "Failed to find config filename.\n");
```

The Message Ressource configuration defines how and to what destinations will be sent.

2.3.1 Special Cases

- `JCR == NULL`: in this case, it is tried to identify the `JCR` automatically. If no `JCR` is found, the message is treated as Daemon message, with fallback to output to `STDOUT`.
- `JCR.JobId == 0` and `JCR.dir_socket != NULL`: a socket connection to the Director exists (normally on the File or Storage Daemon). The message is send directly to the Director via this socket connection.

2.4 Queued Job Messages

Queued Job messages are similar to Jmsg()s except that the message is Queued rather than immediately dispatched. This is necessary within the network subroutines and in the message editing routines. This is to prevent recursive loops, and to ensure that messages can be delivered even in the event of a network error.

```
Qmsg(jcr, M_INFO, 0, "File skipped. Not newer: %s\n", attr->ofname);
```

2.5 Error Messages

Error messages are messages that are related to the daemon as a whole rather than a particular job. For example, an out of memory condition my generate an error message. They should be very rarely needed. In general, you should be using Job and Job Queued messages (Jmsg and Qmsg). They are coded as:

```
EmsgN(ERROR_CODE, 0, message, arg1, ...)
```

As with debug messages, you must explicitly code the of arguments to be substituted in the message.

Some actual examples are:

```
Emsg1(M_ABORT, 0, "Cannot create message thread: %s\n", strerror(status));
Emsg3(M_WARNING, 0, "Connect to File daemon %s at %s:%d failed. Retrying ...\n",␣
↪client->hdr.name, client->address, client->port);
Emsg3(M_FATAL, 0, "Bad response from File daemon to %s command: %d %s\n", cmd, n,␣
↪strerror(errno));
```

In essence, a `EmsgN(ERROR_CODE,0,message,arg1,...)` call resolves to:

```
DmsgN(10, message, arg1, ...)
JmsgN(NULL, ERROR_CODE, 0, message, arg1, ...)
```

CHAPTER

THREE

BAREOS FD PLUGIN API

To write a Bareos plugin, you create a dynamic shared object program (or dll on Win32) with a particular name and two exported entry points, place it in the **Plugins Directory**, which is defined in the **bareos-fd.conf** file in the **Client** resource, and when the FD starts, it will load all the plugins that end with **-fd.so** (or **-fd.dll** on Win32) found in that directory.

3.1 Normal vs Command vs Options Plugins

In general, there are three ways that plugins are called. The first way, is when a particular event is detected in Bareos, it will transfer control to each plugin that is loaded in turn informing the plugin of the event. This is very similar to how a **RunScript** works, and the events are very similar. Once the plugin gets control, it can interact with Bareos by getting and setting Bareos variables. In this way, it behaves much like a RunScript. Currently very few Bareos variables are defined, but they will be implemented as the need arises, and it is very extensible.

We plan to have plugins register to receive events that they normally would not receive, such as an event for each file examined for backup or restore. This feature is not yet implemented.

The second type of plugin, which is more useful and fully implemented in the current version is what we call a command plugin. As with all plugins, it gets notified of important events as noted above (details described below), but in addition, this kind of plugin can accept a command line, which is a:

```
Plugin = <command-string>
```

directive that is placed in the Include section of a FileSet and is very similar to the "File =" directive. When this Plugin directive is encountered by Bareos during backup, it passes the "command" part of the Plugin directive only to the plugin that is explicitly named in the first field of that command string. This allows that plugin to backup any file or files on the system that it wants. It can even create "virtual files" in the catalog that contain data to be restored but do not necessarily correspond to actual files on the filesystem.

The important features of the command plugin entry points are:

- It is triggered by a "Plugin =" directive in the FileSet
- Only a single plugin is called that is named on the "Plugin =" directive.
- The full command string after the "Plugin =" is passed to the plugin so that it can be told what to backup/restore.

The third type of plugin is the Options Plugin, this kind of plugin is useful to implement some custom filter on data. For example, you can implement a compression algorithm in a very simple way. Bareos will call this plugin for each file that is selected in a FileSet (according to Wild/Regex/Exclude/Include rules). As with all plugins, it gets notified of important events as noted above (details described below), but in addition, this kind of plugin can be placed in a Options group, which is a:

```
 FileSet {
    Name = TestFS
    Include {
       Options {
          Compression = GZIP1
          Signature = MD5
          Wild = "*.txt"
          Plugin = <command-string>
       }
       File = /
    }
}
```

3.2 Loading Plugins

Once the File daemon loads the plugins, it asks the OS for the two entry points (loadPlugin and unloadPlugin) then calls the **loadPlugin** entry point (see below).

Bareos passes information to the plugin through this call and it gets back information that it needs to use the plugin. Later, Bareos will call particular functions that are defined by the **loadPlugin** interface.

When Bareos is finished with the plugin (when Bareos is going to exit), it will call the **unloadPlugin** entry point.

The two entry points are:

```
bRC loadPlugin(bInfo *lbinfo, bFuncs *lbfuncs, pInfo **pinfo, pFuncs **pfuncs)

and

bRC unloadPlugin()
```

both these external entry points to the shared object are defined as C entry points to avoid name mangling complications with C++. However, the shared object can actually be written in any language (preferably C or C++) providing that it follows C language calling conventions.

The definitions for **bRC** and the arguments are **src/filed/fd-plugins.h** and so this header file needs to be included in your plugin. It along with **src/lib/plugins.h** define basically the whole plugin interface. Within this header file, it includes the following files:

```
#include <sys/types.h>
#include "config.h"
#include "bc_types.h"
#include "lib/plugins.h"
#include <sys/stat.h>
```

Aside from the **bc_types.h** and **confit.h** headers, the plugin definition uses the minimum code from Bareos. The bc_types.h file is required to ensure that the data type definitions in arguments correspond to the Bareos core code.

The return codes are defined as:

```
typedef enum {
  bRC_OK     = 0,                         /* OK */
  bRC_Stop   = 1,                         /* Stop calling other plugins */
  bRC_Error  = 2,                         /* Some kind of error */
  bRC_More   = 3,                         /* More files to backup */
  bRC_Term   = 4,                         /* Unload me */
```

```
  bRC_Seen  = 5,                        /* Return code from checkFiles */
  bRC_Core  = 6,                        /* Let Bareos core handles this file */
  bRC_Skip  = 7,                        /* Skip the proposed file */
} bRC;
```

At a future point in time, we hope to make the Bareos libbac.a into a shared object so that the plugin can use much more of Bareos's infrastructure, but for this first cut, we have tried to minimize the dependence on Bareos.

3.3 loadPlugin

As previously mentioned, the **loadPlugin** entry point in the plugin is called immediately after Bareos loads the plugin when the File daemon itself is first starting. This entry point is only called once during the execution of the File daemon. In calling the plugin, the first two arguments are information from Bareos that is passed to the plugin, and the last two arguments are information about the plugin that the plugin must return to Bareos. The call is:

```
bRC loadPlugin(bInfo *lbinfo, bFuncs *lbfuncs, pInfo **pinfo, pFuncs **pfuncs)
```

and the arguments are:

lbinfo This is information about Bareos in general. Currently, the only value defined in the bInfo structure is the version, which is the Bareos plugin interface version, currently defined as 1. The **size** is set to the byte size of the structure. The exact definition of the bInfo structure as of this writing is:

```
typedef struct s_bareosInfo {
   uint32_t size;
   uint32_t version;
} bInfo;
```

lbfuncs The bFuncs structure defines the callback entry points within Bareos that the plugin can use register events, get Bareos values, set Bareos values, and send messages to the Job output or debug output.

The exact definition as of this writing is:

```
typedef struct s_bareosFuncs {
   uint32_t size;
   uint32_t version;
   bRC (*registerBareosEvents)(bpContext *ctx, ...);
   bRC (*getBareosValue)(bpContext *ctx, bVariable var, void *value);
   bRC (*setBareosValue)(bpContext *ctx, bVariable var, void *value);
   bRC (*JobMessage)(bpContext *ctx, const char *file, int line,
       int type, utime_t mtime, const char *fmt, ...);
   bRC (*DebugMessage)(bpContext *ctx, const char *file, int line,
       int level, const char *fmt, ...);
   void *(*bareosMalloc)(bpContext *ctx, const char *file, int line,
       size_t size);
   void (*bareosFree)(bpContext *ctx, const char *file, int line, void *mem);
} bFuncs;
```

We will discuss these entry points and how to use them a bit later when describing the plugin code.

pInfo When the loadPlugin entry point is called, the plugin must initialize an information structure about the plugin and return a pointer to this structure to Bareos.

The exact definition as of this writing is:

```
typedef struct s_pluginInfo {
   uint32_t size;
   uint32_t version;
   const char *plugin_magic;
   const char *plugin_license;
   const char *plugin_author;
   const char *plugin_date;
   const char *plugin_version;
   const char *plugin_description;
} pInfo;
```

Where:

version is the current Bareos defined plugin interface version, currently set to 1. If the interface version differs from the current version of Bareos, the plugin will not be run (not yet implemented).

plugin_magic is a pointer to the text string "*FDPluginData*", a sort of sanity check. If this value is not specified, the plugin will not be run (not yet implemented).

plugin_license is a pointer to a text string that describes the plugin license. Bareos will only accept compatible licenses (not yet implemented).

plugin_author is a pointer to the text name of the author of the program. This string can be anything but is generally the author's name.

plugin_date is the pointer text string containing the date of the plugin. This string can be anything but is generally some human readable form of the date.

plugin_version is a pointer to a text string containing the version of the plugin. The contents are determined by the plugin writer.

plugin_description is a pointer to a string describing what the plugin does. The contents are determined by the plugin writer.

The pInfo structure must be defined in static memory because Bareos does not copy it and may refer to the values at any time while the plugin is loaded. All values must be supplied or the plugin will not run (not yet implemented). All text strings must be either ASCII or UTF-8 strings that are terminated with a zero byte.

pFuncs When the loadPlugin entry point is called, the plugin must initialize an entry point structure about the plugin and return a pointer to this structure to Bareos. This structure contains pointer to each of the entry points that the plugin must provide for Bareos. When Bareos is actually running the plugin, it will call the defined entry points at particular times. All entry points must be defined.

The pFuncs structure must be defined in static memory because Bareos does not copy it and may refer to the values at any time while the plugin is loaded.

The exact definition as of this writing is:

```
typedef struct s_pluginFuncs {
   uint32_t size;
   uint32_t version;
   bRC (*newPlugin)(bpContext *ctx);
   bRC (*freePlugin)(bpContext *ctx);
   bRC (*getPluginValue)(bpContext *ctx, pVariable var, void *value);
   bRC (*setPluginValue)(bpContext *ctx, pVariable var, void *value);
   bRC (*handlePluginEvent)(bpContext *ctx, bEvent *event, void *value);
   bRC (*startBackupFile)(bpContext *ctx, struct save_pkt *sp);
   bRC (*endBackupFile)(bpContext *ctx);
   bRC (*startRestoreFile)(bpContext *ctx, const char *cmd);
   bRC (*endRestoreFile)(bpContext *ctx);
   bRC (*pluginIO)(bpContext *ctx, struct io_pkt *io);
```

```
   bRC (*createFile)(bpContext *ctx, struct restore_pkt *rp);
   bRC (*setFileAttributes)(bpContext *ctx, struct restore_pkt *rp);
   bRC (*checkFile)(bpContext *ctx, char *fname);
} pFuncs;
```

The details of the entry points will be presented in separate sections below.

Where:

size is the byte size of the structure.

version is the plugin interface version currently set to 3.

Sample code for loadPlugin:

```
bfuncs = lbfuncs;                  /* set Bareos funct pointers */
binfo  = lbinfo;
*pinfo  = &pluginInfo;             /* return pointer to our info */
*pfuncs = &pluginFuncs;            /* return pointer to our functions */

 return bRC_OK;
```

where pluginInfo and pluginFuncs are statically defined structures. See bpipe-fd.c for details.

3.4 Plugin Entry Points

This section will describe each of the entry points (subroutines) within the plugin that the plugin must provide for Bareos, when they are called and their arguments. As noted above, pointers to these subroutines are passed back to Bareos in the pFuncs structure when Bareos calls the loadPlugin() externally defined entry point.

3.4.1 newPlugin(bpContext *ctx)

This is the entry point that Bareos will call when a new "instance" of the plugin is created. This typically happens at the beginning of a Job. If 10 Jobs are running simultaneously, there will be at least 10 instances of the plugin.

The bpContext structure will be passed to the plugin, and during this call, if the plugin needs to have its private working storage that is associated with the particular instance of the plugin, it should create it from the heap (malloc the memory) and store a pointer to its private working storage in the **pContext** variable. Note: since Bareos is a multi-threaded program, you must not keep any variable data in your plugin unless it is truly meant to apply globally to the whole plugin. In addition, you must be aware that except the first and last call to the plugin (loadPlugin and unloadPlugin) all the other calls will be made by threads that correspond to a Bareos job. The bpContext that will be passed for each thread will remain the same throughout the Job thus you can keep your private Job specific data in it (**bContext**).

```
typedef struct s_bpContext {
  void *pContext;   /* Plugin private context */
  void *bContext;   /* Bareos private context */
} bpContext;
```

This context pointer will be passed as the first argument to all the entry points that Bareos calls within the plugin. Needless to say, the plugin should not change the bContext variable, which is Bareos's private context pointer for this instance (Job) of this plugin.

3.4.2 freePlugin(bpContext *ctx)

This entry point is called when the this instance of the plugin is no longer needed (the Job is ending), and the plugin should release all memory it may have allocated for this particular instance (Job) i.e. the pContext. This is not the final termination of the plugin signaled by a call to **unloadPlugin**. Any other instances (Job) will continue to run, and the entry point **newPlugin** may be called again if other jobs start.

3.4.3 getPluginValue(bpContext *ctx, pVariable var, void *value)

Bareos will call this entry point to get a value from the plugin. This entry point is currently not called.

3.4.4 setPluginValue(bpContext *ctx, pVariable var, void *value)

Bareos will call this entry point to set a value in the plugin. This entry point is currently not called.

3.4.5 handlePluginEvent(bpContext *ctx, bEvent *event, void *value)

This entry point is called when Bareos encounters certain events (discussed below). This is, in fact, the main way that most plugins get control when a Job runs and how they know what is happening in the job. It can be likened to the **RunScript** feature that calls external programs and scripts, and is very similar to the Bareos Python interface. When the plugin is called, Bareos passes it the pointer to an event structure (bEvent), which currently has one item, the eventType:

```
typedef struct s_bEvent {
   uint32_t eventType;
} bEvent;
```

which defines what event has been triggered, and for each event, Bareos will pass a pointer to a value associated with that event. If no value is associated with a particular event, Bareos will pass a NULL pointer, so the plugin must be careful to always check value pointer prior to dereferencing it.

The current list of events are:

```
typedef enum {
  bEventJobStart                    = 1,
  bEventJobEnd                      = 2,
  bEventStartBackupJob              = 3,
  bEventEndBackupJob                = 4,
  bEventStartRestoreJob             = 5,
  bEventEndRestoreJob               = 6,
  bEventStartVerifyJob              = 7,
  bEventEndVerifyJob                = 8,
  bEventBackupCommand               = 9,
  bEventRestoreCommand              = 10,
  bEventLevel                       = 11,
  bEventSince                       = 12,
  bEventCancelCommand               = 13,  /* Executed by another thread */

  /* Just before bEventVssPrepareSnapshot */
  bEventVssBackupAddComponents      = 14,

  bEventVssRestoreLoadComponentMetadata = 15,
  bEventVssRestoreSetComponentsSelected = 16,
  bEventRestoreObject               = 17,
```

```
  bEventEndFileSet                    = 18,
  bEventPluginCommand                 = 19,
  bEventVssBeforeCloseRestore         = 21,

  /* Add drives to VSS snapshot
   *  argument: char[27] drivelist
   * You need to add them without duplicates,
   * see fd_common.h add_drive() copy_drives() to get help
   */
  bEventVssPrepareSnapshot            = 22,
  bEventOptionPlugin                  = 23,
  bEventHandleBackupFile              = 24 /* Used with Options Plugin */

} bEventType;
```

Most of the above are self-explanatory.

bEventJobStart is called whenever a Job starts. The value passed is a pointer to a string that contains: "Jobid=nnn Job=job-name". Where nnn will be replaced by the JobId and job-name will be replaced by the Job name. The variable is temporary so if you need the values, you must copy them.

bEventJobEnd is called whenever a Job ends. No value is passed.

bEventStartBackupJob is called when a Backup Job begins. No value is passed.

bEventEndBackupJob is called when a Backup Job ends. No value is passed.

bEventStartRestoreJob is called when a Restore Job starts. No value is passed.

bEventEndRestoreJob is called when a Restore Job ends. No value is passed.

bEventStartVerifyJob is called when a Verify Job starts. No value is passed.

bEventEndVerifyJob is called when a Verify Job ends. No value is passed.

bEventBackupCommand is called prior to the bEventStartBackupJob and the plugin is passed the command string (everything after the equal sign in "Plugin =" as the value.

Note, if you intend to backup a file, this is an important first point to write code that copies the command string passed into your pContext area so that you will know that a backup is being performed and you will know the full contents of the "Plugin =" command (i.e. what to backup and what virtual filename the user wants to call it.

bEventRestoreCommand is called prior to the bEventStartRestoreJob and the plugin is passed the command string (everything after the equal sign in "Plugin =" as the value.

See the notes above concerning backup and the command string. This is the point at which Bareos passes you the original command string that was specified during the backup, so you will want to save it in your pContext area for later use when Bareos calls the plugin again.

bEventLevel is called when the level is set for a new Job. The value is a 32 bit integer stored in the void*, which represents the Job Level code.

bEventSince is called when the since time is set for a new Job. The value is a time_t time at which the last job was run.

bEventCancelCommand is called whenever the currently running Job is cancelled. Be warned that this event is sent by a different thread.

bEventVssBackupAddComponents bEventPluginCommand is called for each PluginCommand present in the current FileSet. The event will be sent only on plugin specifed in the command. The argument is the PluginCommand (not valid after the call).

bEventHandleBackupFile is called for each file of a FileSet when using a Options Plugin. If the plugin returns CF_OK, it will be used for the backup, if it returns CF_SKIP, the file will be skipped. Anything else will backup the file with Bareos core functions.

During each of the above calls, the plugin receives either no specific value or only one value, which in some cases may not be sufficient. However, knowing the context of the event, the plugin can call back to the Bareos entry points it was passed during the **loadPlugin** call and get to a number of Bareos variables. (at the current time few Bareos variables are implemented, but it easily extended at a future time and as needs require).

3.4.6 startBackupFile(bpContext *ctx, struct save_pkt *sp)

This entry point is called only if your plugin is a command plugin, and it is called when Bareos encounters the "Plugin =" directive in the Include section of the FileSet. Called when beginning the backup of a file. Here Bareos provides you with a pointer to the **save_pkt** structure and you must fill in this packet with the "attribute" data of the file.

```
struct save_pkt {
   int32_t pkt_size;                  /* size of this packet */
   char *fname;                       /* Full path and filename */
   char *link;                        /* Link name if any */
   struct stat statp;                 /* System stat() packet for file */
   int32_t type;                      /* FT_xx for this file */
   uint32_t flags;                    /* Bareos internal flags */
   bool portable;                     /* set if data format is portable */
   char *cmd;                         /* command */
   uint32_t delta_seq;                /* Delta sequence number */
   char *object_name;                 /* Object name to create */
   char *object;                      /* restore object data to save */
   int32_t object_len;                /* restore object length */
   int32_t index;                     /* restore object index */
   int32_t pkt_end;                   /* end packet sentinel */
};
```

The second argument is a pointer to the **save_pkt** structure for the file to be backed up. The plugin is responsible for filling in all the fields of the **save_pkt**. If you are backing up a real file, then generally, the statp structure can be filled in by doing a **stat** system call on the file.

If you are backing up a database or something that is more complex, you might want to create a virtual file. That is a file that does not actually exist on the filesystem, but represents say an object that you are backing up. In that case, you need to ensure that the **fname** string that you pass back is unique so that it does not conflict with a real file on the system, and you need to artifically create values in the statp packet.

Example programs such as **bpipe-fd.c** show how to set these fields. You must take care not to store pointers the stack in the pointer fields such as fname and link, because when you return from your function, your stack entries will be destroyed. The solution in that case is to malloc() and return the pointer to it. In order to not have memory leaks, you should store a pointer to all memory allocated in your pContext structure so that in subsequent calls or at termination, you can release it back to the system.

Once the backup has begun, Bareos will call your plugin at the **pluginIO** entry point to "read" the data to be backed up. Please see the **bpipe-fd.c** plugin for how to do I/O.

Example of filling in the save_pkt as used in bpipe-fd.c:

```
struct plugin_ctx *p_ctx = (struct plugin_ctx *)ctx->pContext;
time_t now = time(NULL);
sp->fname = p_ctx->fname;
sp->statp.st_mode = 0700 | S_IFREG;
sp->statp.st_ctime = now;
sp->statp.st_mtime = now;
```

```
sp->statp.st_atime = now;
sp->statp.st_size = -1;
sp->statp.st_blksize = 4096;
sp->statp.st_blocks = 1;
p_ctx->backup = true;
return bRC_OK;
```

Note: the filename to be created has already been created from the command string previously sent to the plugin and is in the plugin context (p_ctx->fname) and is a malloc()ed string. This example creates a regular file (S_IFREG), with various fields being created.

In general, the sequence of commands issued from Bareos to the plugin to do a backup while processing the "Plugin =" directive are:

1. generate a bEventBackupCommand event to the specified plugin and pass it the command string.
2. make a startPluginBackup call to the plugin, which fills in the data needed in save_pkt to save as the file attributes and to put on the Volume and in the catalog.
3. call Bareos's internal save_file() subroutine to save the specified file. The plugin will then be called at pluginIO() to "open" the file, and then to read the file data. Note, if you are dealing with a virtual file, the "open" operation is something the plugin does internally and it doesn't necessarily mean opening a file on the filesystem. For example in the case of the bpipe-fd.c program, it initiates a pipe to the requested program. Finally when the plugin signals to Bareos that all the data was read, Bareos will call the plugin with the "close" pluginIO() function.

3.4.7 endBackupFile(bpContext *ctx)

Called at the end of backing up a file for a command plugin. If the plugin's work is done, it should return bRC_OK. If the plugin wishes to create another file and back it up, then it must return bRC_More (not yet implemented). This is probably a good time to release any malloc()ed memory you used to pass back filenames.

3.4.8 startRestoreFile(bpContext *ctx, const char *cmd)

Called when the first record is read from the Volume that was previously written by the command plugin.

3.4.9 createFile(bpContext *ctx, struct restore_pkt *rp)

Called for a command plugin to create a file during a Restore job before restoring the data. This entry point is called before any I/O is done on the file. After this call, Bareos will call pluginIO() to open the file for write.

The data in the restore_pkt is passed to the plugin and is based on the data that was originally given by the plugin during the backup and the current user restore settings (e.g. where, RegexWhere, replace). This allows the plugin to first create a file (if necessary) so that the data can be transmitted to it. The next call to the plugin will be a pluginIO command with a request to open the file write-only.

This call must return one of the following values:

```
 enum {
   CF_SKIP = 1,       /* skip file (not newer or something) */
   CF_ERROR,          /* error creating file */
   CF_EXTRACT,        /* file created, data to extract */
   CF_CREATED,        /* file created, no data to extract */
   CF_CORE            /* let bareos core handles the file creation */
};
```

in the restore_pkt value **create_status**. For a normal file, unless there is an error, you must return **CF_EXTRACT**.

```
struct restore_pkt {
   int32_t pkt_size;                  /* size of this packet */
   int32_t stream;                    /* attribute stream id */
   int32_t data_stream;               /* id of data stream to follow */
   int32_t type;                      /* file type FT */
   int32_t file_index;                /* file index */
   int32_t LinkFI;                    /* file index to data if hard link */
   uid_t uid;                         /* userid */
   struct stat statp;                 /* decoded stat packet */
   const char *attrEx;                /* extended attributes if any */
   const char *ofname;                /* output filename */
   const char *olname;                /* output link name */
   const char *where;                 /* where */
   const char *RegexWhere;            /* regex where */
   int replace;                       /* replace flag */
   int create_status;                 /* status from createFile() */
   int32_t pkt_end;                   /* end packet sentinel */

};
```

Typical code to create a regular file would be the following:

```
struct plugin_ctx *p_ctx = (struct plugin_ctx *)ctx->pContext;
time_t now = time(NULL);
sp->fname = p_ctx->fname;   /* set the full path/filename I want to create */
sp->type = FT_REG;
sp->statp.st_mode = 0700 | S_IFREG;
sp->statp.st_ctime = now;
sp->statp.st_mtime = now;
sp->statp.st_atime = now;
sp->statp.st_size = -1;
sp->statp.st_blksize = 4096;
sp->statp.st_blocks = 1;
return bRC_OK;
```

This will create a virtual file. If you are creating a file that actually exists, you will most likely want to fill the statp packet using the stat() system call.

Creating a directory is similar, but requires a few extra steps:

```
struct plugin_ctx *p_ctx = (struct plugin_ctx *)ctx->pContext;
time_t now = time(NULL);
sp->fname = p_ctx->fname;   /* set the full path I want to create */
sp->link = xxx; where xxx is p_ctx->fname with a trailing forward slash
sp->type = FT_DIREND
sp->statp.st_mode = 0700 | S_IFDIR;
sp->statp.st_ctime = now;
sp->statp.st_mtime = now;
sp->statp.st_atime = now;
sp->statp.st_size = -1;
sp->statp.st_blksize = 4096;
sp->statp.st_blocks = 1;
return bRC_OK;
```

The link field must be set with the full cononical path name, which always ends with a forward slash. If you do not terminate it with a forward slash, you will surely have problems later.

As with the example that creates a file, if you are backing up a real directory, you will want to do an stat() on the directory.

Note, if you want the directory permissions and times to be correctly restored, you must create the directory **after** all the file directories have been sent to Bareos. That allows the restore process to restore all the files in a directory using default directory options, then at the end, restore the directory permissions. If you do it the other way around, each time you restore a file, the OS will modify the time values for the directory entry.

3.4.10 setFileAttributes(bpContext *ctx, struct restore_pkt *rp)

This is call not yet implemented. Called for a command plugin.

See the definition of **restre_pkt** in the above section.

3.4.11 endRestoreFile(bpContext *ctx)

Called when a command plugin is done restoring a file.

3.4.12 pluginIO(bpContext *ctx, struct io_pkt *io)

Called to do the input (backup) or output (restore) of data from or to a file for a command plugin. These routines simulate the Unix read(), write(), open(), close(), and lseek() I/O calls, and the arguments are passed in the packet and the return values are also placed in the packet. In addition for Win32 systems the plugin must return two additional values (described below).

```
 enum {
   IO_OPEN = 1,
   IO_READ = 2,
   IO_WRITE = 3,
   IO_CLOSE = 4,
   IO_SEEK = 5
};

struct io_pkt {
   int32_t pkt_size;                  /* Size of this packet */
   int32_t func;                      /* Function code */
   int32_t count;                     /* read/write count */
   mode_t mode;                       /* permissions for created files */
   int32_t flags;                     /* Open flags */
   char *buf;                         /* read/write buffer */
   const char *fname;                 /* open filename */
   int32_t status;                    /* return status */
   int32_t io_errno;                  /* errno code */
   int32_t lerror;                    /* Win32 error code */
   int32_t whence;                    /* lseek argument */
   boffset_t offset;                  /* lseek argument */
   bool win32;                        /* Win32 GetLastError returned */
   int32_t pkt_end;                   /* end packet sentinel */
};
```

The particular Unix function being simulated is indicated by the **func**, which will have one of the IO_OPEN, IO_READ, ... codes listed above. The status code that would be returned from a Unix call is returned in **status** for IO_OPEN, IO_CLOSE, IO_READ, and IO_WRITE. The return value for IO_SEEK is returned in **offset** which in general is a 64 bit value.

When there is an error on Unix systems, you must always set io_error, and on a Win32 system, you must always set win32, and the returned value from the OS call GetLastError() in lerror.

For all except IO_SEEK, **status** is the return result. In general it is a positive integer unless there is an error in which case it is -1.

The following describes each call and what you get and what you should return:

IO_OPEN You will be passed fname, mode, and flags. You must set on return: status, and if there is a Unix error io_errno must be set to the errno value, and if there is a Win32 error win32 and lerror.

IO_READ You will be passed: count, and buf (buffer of size count). You must set on return: status to the number of bytes read into the buffer (buf) or -1 on an error, and if there is a Unix error io_errno must be set to the errno value, and if there is a Win32 error, win32 and lerror must be set.

IO_WRITE You will be passed: count, and buf (buffer of size count). You must set on return: status to the number of bytes written from the buffer (buf) or -1 on an error, and if there is a Unix error io_errno must be set to the errno value, and if there is a Win32 error, win32 and lerror must be set.

IO_CLOSE Nothing will be passed to you. On return you must set status to 0 on success and -1 on failure. If there is a Unix error io_errno must be set to the errno value, and if there is a Win32 error, win32 and lerror must be set.

IO_LSEEK You will be passed: offset, and whence. offset is a 64 bit value and is the position to seek to relative to whence. whence is one of the following SEEK_SET, SEEK_CUR, or SEEK_END indicating to either to seek to an absolute possition, relative to the current position or relative to the end of the file. You must pass back in offset the absolute location to which you seeked. If there is an error, offset should be set to -1. If there is a Unix error io_errno must be set to the errno value, and if there is a Win32 error, win32 and lerror must be set.

Note: Bareos will call IO_SEEK only when writing a sparse file.

3.4.13 bool checkFile(bpContext *ctx, char *fname)

If this entry point is set, Bareos will call it after backing up all file data during an Accurate backup. It will be passed the full filename for each file that Bareos is proposing to mark as deleted. Only files previously backed up but not backed up in the current session will be marked to be deleted. If you return **false**, the file will be be marked deleted. If you return **true** the file will not be marked deleted. This permits a plugin to ensure that previously saved virtual files or files controlled by your plugin that have not change (not backed up in the current job) are not marked to be deleted. This entry point will only be called during Accurate Incrmental and Differential backup jobs.

3.5 Bareos Plugin Entrypoints

When Bareos calls one of your plugin entrypoints, you can call back to the entrypoints in Bareos that were supplied during the xxx plugin call to get or set information within Bareos.

3.5.1 bRC registerBareosEvents(bpContext *ctx, . . .)

This Bareos entrypoint will allow you to register to receive events that are not autmatically passed to your plugin by default. This entrypoint currently is unimplemented.

3.5.2 bRC getBareosValue(bpContext *ctx, bVariable var, void *value)

Calling this entrypoint, you can obtain specific values that are available in Bareos. The following Variables can be referenced:

- bVarJobId returns an int
- bVarFDName returns a char *
- bVarLevel returns an int
- bVarClient returns a char *
- bVarJobName returns a char *
- bVarJobStatus returns an int
- bVarSinceTime returns an int (time_t)
- bVarAccurate returns an int

3.5.3 bRC setBareosValue(bpContext *ctx, bVariable var, void *value)

Calling this entrypoint allows you to set particular values in Bareos. The only variable that can currently be set is **bVarFileSeen** and the value passed is a char * that points to the full filename for a file that you are indicating has been seen and hence is not deleted.

3.5.4 bRC JobMessage(bpContext *ctx, const char *file, int line, int type, utime_t mtime, const char *fmt, ...)

This call permits you to put a message in the Job Report.

3.5.5 bRC DebugMessage(bpContext *ctx, const char *file, int line, int level, const char *fmt, ...)

This call permits you to print a debug message.

3.5.6 void bareosMalloc(bpContext *ctx, const char *file, int line, size_t size)

This call permits you to obtain memory from Bareos's memory allocator.

3.5.7 void bareosFree(bpContext *ctx, const char *file, int line, void *mem)

This call permits you to free memory obtained from Bareos's memory allocator.

3.6 Building Bareos Plugins

There is currently one sample program **example-plugin-fd.c** and one working plugin **bpipe-fd.c** that can be found in the Bareos **src/plugins/fd** directory. Both are built with the following:

```
cd <bareos-source>
./configure <your-options>
make
...
cd src/plugins/fd
make
make test
```

After building Bareos and changing into the src/plugins/fd directory, the **make** command will build the **bpipe-fd.so** plugin, which is a very useful and working program.

The **make test** command will build the **example-plugin-fd.so** plugin and a binary named **main**, which is build from the source code located in **src/filed/fd_plugins.c**.

If you execute **./main**, it will load and run the example-plugin-fd plugin simulating a small number of the calling sequences that Bareos uses in calling a real plugin. This allows you to do initial testing of your plugin prior to trying it with Bareos.

You can get a good idea of how to write your own plugin by first studying the example-plugin-fd, and actually running it. Then it can also be instructive to read the bpipe-fd.c code as it is a real plugin, which is still rather simple and small.

When actually writing your own plugin, you may use the example-plugin-fd.c code as a template for your code.

CHAPTER

FOUR

PLATFORM SUPPORT

4.1 General

This chapter describes the requirements for having a supported platform (Operating System). In general, Bareos is quite portable. It supports 32 and 64 bit architectures as well as bigendian and littleendian machines. For full support, the platform (Operating System) should implement POSIX Unix system calls. However, for File daemon support only, a small compatibility library can be written to support almost any architecture.

4.2 Requirements to become a Supported Platform

As mentioned above, in order to become a fully supported platform, it must support POSIX Unix system calls. In addition, the following requirements must be met:

CHAPTER

FIVE

DAEMON PROTOCOL

5.1 General

This document describes the protocols used between the various daemons. As Bareos has developed, it has become quite out of date. The general idea still holds true, but the details of the fields for each command, and indeed the commands themselves have changed considerably.

It is intended to be a technical discussion of the general daemon protocols and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bareos**.

5.2 Low Level Network Protocol

At the lowest level, the network protocol is handled by **BSOCK** packets which contain a lot of information about the status of the network connection: who is at the other end, etc. Each basic **Bareos** network read or write actually consists of two low level network read/writes. The first write always sends four bytes of data in machine independent byte order. If data is to follow, the first four bytes are a positive non-zero integer indicating the length of the data that follow in the subsequent write. If the four byte integer is zero or negative, it indicates a special request, a sort of network signaling capability. In this case, no data packet will follow. The low level BSOCK routines expect that only a single thread is accessing the socket at a time. It is advised that multiple threads do not read/write the same socket. If you must do this, you must provide some sort of locking mechanism. It would not be appropriate for efficiency reasons to make every call to the BSOCK routines lock and unlock the packet.

5.3 General Daemon Protocol

In general, all the daemons follow the following global rules. There may be exceptions depending on the specific case. Normally, one daemon will be sending commands to another daemon (specifically, the Director to the Storage daemon and the Director to the File daemon).

- Commands are always ASCII commands that are upper/lower case dependent as well as space sensitive.
- All binary data is converted into ASCII (either with printf statements or using base64 encoding).
- All responses to commands sent are always prefixed with a return numeric code where codes in the 1000's are reserved for the Director, the 2000's are reserved for the File daemon, and the 3000's are reserved for the Storage daemon.
- Any response that is not prefixed with a numeric code is a command (or subcommand if you like) coming from the other end. For example, while the Director is corresponding with the Storage daemon, the Storage daemon can request Catalog services from the Director. This convention permits each side to send commands to the other daemon while simultaneously responding to commands.

- Any response that is of zero length, depending on the context, either terminates the data stream being sent or terminates command mode prior to closing the connection.
- Any response that is of negative length is a special sign that normally requires a response. For example, during data transfer from the File daemon to the Storage daemon, normally the File daemon sends continuously without intervening reads. However, periodically, the File daemon will send a packet of length -1 indicating that the current data stream is complete and that the Storage daemon should respond to the packet with an OK, ABORT JOB, PAUSE, etc. This permits the File daemon to efficiently send data while at the same time occasionally "polling" the Storage daemon for his status or any special requests.

  Currently, these negative lengths are specific to the daemon, but shortly, the range 0 to -999 will be standard daemon wide signals, while -1000 to -1999 will be for Director user, -2000 to -2999 for the File daemon, and -3000 to -3999 for the Storage daemon.

5.4 The Protocol Used Between the Director and the Storage Daemon

Before sending commands to the File daemon, the Director opens a Message channel with the Storage daemon, identifies itself and presents its password. If the password check is OK, the Storage daemon accepts the Director. The Director then passes the Storage daemon, the JobId to be run as well as the File daemon authorization (append, read all, or read for a specific session). The Storage daemon will then pass back to the Director a enabling key for this JobId that must be presented by the File daemon when opening the job. Until this process is complete, the Storage daemon is not available for use by File daemons.

```
SD: listens
DR: makes connection
DR: Hello <Director-name> calling <password>
SD: 3000 OK Hello
DR: JobId=nnn Allow=(append, read) Session=(*, SessionId)
                    (Session not implemented yet)
SD: 3000 OK Job Authorization=<password>
DR: use device=<device-name> media_type=<media-type>
        pool_name=<pool-name> pool_type=<pool_type>
SD: 3000 OK use device
```

For the Director to be authorized, the <Director-name> and the <password> must match the values in one of the Storage daemon's Director resources (there may be several Directors that can access a single Storage daemon).

5.5 The Protocol Used Between the Director and the File Daemon

A typical conversation might look like the following:

```
FD: listens
DR: makes connection
DR: Hello <Director-name> calling <password>
FD: 2000 OK Hello
DR: JobId=nnn Authorization=<password>
FD: 2000 OK Job
DR: storage address = <Storage daemon address> port = <port-number>
          name = <DeviceName> mediatype = <MediaType>
FD: 2000 OK storage
DR: include
DR: <directory1>
DR: <directory2>
    ...
```

```
DR: Null packet
FD: 2000 OK include
DR: exclude
DR: <directory1>
DR: <directory2>
    ...
DR: Null packet
FD: 2000 OK exclude
DR: full
FD: 2000 OK full
DR: save
FD: 2000 OK save
FD: Attribute record for each file as sent to the
    Storage daemon (described above).
FD: Null packet
FD: <append close responses from Storage daemon>
    e.g.
    3000 OK Volumes = <number of volumes>
    3001 Volume = <volume-id> <start file> <start block>
         <end file> <end block> <volume session-id>
    3002 Volume data = <date/time of last write> <Number bytes written>
         <number errors>
    ... additional Volume / Volume data pairs for volumes 2 .. n
FD: Null packet
FD: close socket
```

5.6 The Save Protocol Between the File Daemon and the Storage Daemon

Once the Director has send a **save** command to the File daemon, the File daemon will contact the Storage daemon to begin the save.

In what follows: FD: refers to information set via the network from the File daemon to the Storage daemon, and SD: refers to information set from the Storage daemon to the File daemon.

5.6.1 Command and Control Information

Command and control information is exchanged in human readable ASCII commands.

```
FD: listens
SD: makes connection
FD: append open session = <JobId> [<password>]
SD: 3000 OK ticket = <number>
FD: append data <ticket-number>
SD: 3000 OK data address = <IPaddress> port = <port>
```

5.6.2 Data Information

The Data information consists of the file attributes and data to the Storage daemon. For the most part, the data information is sent one way: from the File daemon to the Storage daemon. This allows the File daemon to transfer information as fast as possible without a lot of handshaking and network overhead.

However, from time to time, the File daemon needs to do a sort of checkpoint of the situation to ensure that everything is going well with the Storage daemon. To do so, the File daemon sends a packet with a negative length indicating that he wishes the Storage daemon to respond by sending a packet of information to the File daemon. The File daemon then waits to receive a packet from the Storage daemon before continuing.

All data sent are in binary format except for the header packet, which is in ASCII. There are two packet types used data transfer mode: a header packet, the contents of which are known to the Storage daemon, and a data packet, the contents of which are never examined by the Storage daemon.

The first data packet to the Storage daemon will be an ASCII header packet consisting of the following data.

<File-Index> <Stream-Id> <Info> where **<File-Index>** is a sequential number beginning from one that increments with each file (or directory) sent.

where **<Stream-Id>** will be 1 for the Attributes record and 2 for uncompressed File data. 3 is reserved for the MD5 signature for the file.

where **<Info>** transmit information about the Stream to the Storage Daemon. It is a character string field where each character has a meaning. The only character currently defined is 0 (zero), which is simply a place holder (a no op). In the future, there may be codes indicating compressed data, encrypted data, etc.

Immediately following the header packet, the Storage daemon will expect any number of data packets. The series of data packets is terminated by a zero length packet, which indicates to the Storage daemon that the next packet will be another header packet. As previously mentioned, a negative length packet is a request for the Storage daemon to temporarily enter command mode and send a reply to the File daemon. Thus an actual conversation might contain the following exchanges:

```
FD: <1 1 0> (header packet)
FD: <data packet containing file-attributes>
FD: Null packet
FD: <1 2 0>
FD: <multiple data packets containing the file data>
FD: Packet length = -1
SD: 3000 OK
FD: <2 1 0>
FD: <data packet containing file-attributes>
FD: Null packet
FD: <2 2 0>
FD: <multiple data packets containing the file data>
FD: Null packet
FD: Null packet
FD: append end session <ticket-number>
SD: 3000 OK end
FD: append close session <ticket-number>
SD: 3000 OK Volumes = <number of volumes>
SD: 3001 Volume = <volumeid> <start file> <start block>
     <end file> <end block> <volume session-id>
SD: 3002 Volume data = <date/time of last write> <Number bytes written>
     <number errors>
SD: ... additional Volume / Volume data pairs for
     volumes 2 .. n
FD: close socket
```

The information returned to the File daemon by the Storage daemon in response to the **append close session** is transmit in turn to the Director.

CHAPTER

SIX

FILE SERVICES DAEMON

Please note, this section is somewhat out of date as the code has evolved significantly. The basic idea has not changed though.

This chapter is intended to be a technical discussion of the File daemon services and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bareos**.

The **Bareos File Services** consist of the programs that run on the system to be backed up and provide the interface between the Host File system and Bareos – in particular, the Director and the Storage services.

When time comes for a backup, the Director gets in touch with the File daemon on the client machine and hands it a set of "marching orders" which, if written in English, might be something like the following:

OK, **File daemon**, it's time for your daily incremental backup. I want you to get in touch with the Storage daemon on host archive.mysite.com and perform the following save operations with the designated options. You'll note that I've attached include and exclude lists and patterns you should apply when backing up the file system. As this is an incremental backup, you should save only files modified since the time you started your last backup which, as you may recall, was 2000-11-19-06:43:38. Please let me know when you're done and how it went. Thank you.

So, having been handed everything it needs to decide what to dump and where to store it, the File daemon doesn't need to have any further contact with the Director until the backup is complete providing there are no errors. If there are errors, the error messages will be delivered immediately to the Director. While the backup is proceeding, the File daemon will send the file coordinates and data for each file being backed up to the Storage daemon, which will in turn pass the file coordinates to the Director to put in the catalog.

During a **Verify** of the catalog, the situation is different, since the File daemon will have an exchange with the Director for each file, and will not contact the Storage daemon.

A **Restore** operation will be very similar to the **Backup** except that during the **Restore** the Storage daemon will not send storage coordinates to the Director since the Director presumably already has them. On the other hand, any error messages from either the Storage daemon or File daemon will normally be sent directly to the Directory (this, of course, depends on how the Message resource is defined).

6.1 Commands Received from the Director for a Backup

To be written ...

6.2 Commands Received from the Director for a Restore

To be written ...

CHAPTER

SEVEN

STORAGE DAEMON DESIGN

This chapter is intended to be a technical discussion of the Storage daemon services and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bareos**.

This document is somewhat out of date.

7.1 SD Design Introduction

The Bareos Storage daemon provides storage resources to a Bareos installation. An individual Storage daemon is associated with a physical permanent storage device (for example, a tape drive, CD writer, tape changer or jukebox, etc.), and may employ auxiliary storage resources (such as space on a hard disk file system) to increase performance and/or optimize use of the permanent storage medium.

Any number of storage daemons may be run on a given machine; each associated with an individual storage device connected to it, and Bareos operations may employ storage daemons on any number of hosts connected by a network, local or remote. The ability to employ remote storage daemons (with appropriate security measures) permits automatic off-site backup, possibly to publicly available backup repositories.

7.2 SD Development Outline

In order to provide a high performance backup and restore solution that scales to very large capacity devices and networks, the storage daemon must be able to extract as much performance from the storage device and network with which it interacts. In order to accomplish this, storage daemons will eventually have to sacrifice simplicity and painless portability in favor of techniques which improve performance. My goal in designing the storage daemon protocol and developing the initial prototype storage daemon is to provide for these additions in the future, while implementing an initial storage daemon which is very simple and portable to almost any POSIX-like environment. This original storage daemon (and its evolved descendants) can serve as a portable solution for non-demanding backup requirements (such as single servers of modest size, individual machines, or small local networks), while serving as the starting point for development of higher performance configurable derivatives which use techniques such as POSIX threads, shared memory, asynchronous I/O, buffering to high-speed intermediate media, and support for tape changers and jukeboxes.

7.3 SD Connections and Sessions

A client connects to a storage server by initiating a conventional TCP connection. The storage server accepts the connection unless its maximum number of connections has been reached or the specified host is not granted access to the storage server. Once a connection has been opened, the client may make any number of Query requests, and/or

initiate (if permitted), one or more Append sessions (which transmit data to be stored by the storage daemon) and/or Read sessions (which retrieve data from the storage daemon).

Most requests and replies sent across the connection are simple ASCII strings, with status replies prefixed by a four digit status code for easier parsing. Binary data appear in blocks stored and retrieved from the storage. Any request may result in a single-line status reply of "3201 Notificationpending", which indicates the client must send a "Query notification" request to retrieve one or more notifications posted to it. Once the notifications have been returned, the client may then resubmit the request which resulted in the 3201 status.

The following descriptions omit common error codes, yet to be defined, which can occur from most or many requests due to events like media errors, restarting of the storage daemon, etc. These details will be filled in, along with a comprehensive list of status codes along with which requests can produce them in an update to this document.

7.3.1 SD Append Requests

append open session = <JobId> [<Password>] A data append session is opened with the Job ID given by *JobId* with client password (if required) given by *Password*. If the session is successfully opened, a status of 3000 OK is returned with a "ticket = *number*" reply used to identify subsequent messages in the session. If too many sessions are open, or a conflicting session (for example, a read in progress when simultaneous read and append sessions are not permitted), a status of "3502 Volume busy" is returned. If no volume is mounted, or the volume mounted cannot be appended to, a status of "3503 Volume not mounted" is returned.

append data = <ticket-number> If the append data is accepted, a status of 3000 OK data address = <IPaddress> port = <port> is returned, where the IPaddress and port specify the IP address and port number of the data channel. Error status codes are 3504 Invalid ticket number and 3505 Session aborted, the latter of which indicates the entire append session has failed due to a daemon or media error.

Once the File daemon has established the connection to the data channel opened by the Storage daemon, it will transfer a header packet followed by any number of data packets. The header packet is of the form:

file-index> <stream-id> <info>

The details are specified in the section of this document.

***append abort session = <ticket-number>** The open append session with ticket *ticket-number* is aborted; any blocks not yet written to permanent media are discarded. Subsequent attempts to append data to the session will receive an error status of 3505Session aborted.

append end session = <ticket-number> The open append session with ticket *ticket-number* is marked complete; no further blocks may be appended. The storage daemon will give priority to saving any buffered blocks from this session to permanent media as soon as possible.

append close session = <ticket-number> The append session with ticket *ticket* is closed. This message does not receive an 3000 OK reply until all of the content of the session are stored on permanent media, at which time said reply is given, followed by a list of volumes, from first to last, which contain blocks from the session, along with the first and last file and block on each containing session data and the volume session key identifying data from that session in lines with the following format:

Volume-id> <start-file> <start-block> <end-file> <end-block> <volume-session-id>where *Volume-id* is the volume label, *start-file* and *start-block* are the file and block containing the first data from that session on the volume, *end-file* and *end-block* are the file and block with the last data from the session on the volume and *volume-session-id* is the volume session ID for blocks from the session stored on that volume.

7.3.2 SD Read Requests

Read open session = <JobId> <Volume-id> <start-file> <start-block> <end-file> <end-block> <volume-session-id> <password> where *Volume-id* is the volume label, *start-file* and *start-block* are the file and block containing the first data

from that session on the volume, *end-file* and *end-block* are the file and block with the last data from the session on the volume and *volume-session-id* is the volume session ID for blocks from the session stored on that volume.

If the session is successfully opened, a status of

‘‘

is returned with a reply used to identify subsequent messages in the session. If too many sessions are open, or a conflicting session (for example, an append in progress when simultaneous read and append sessions are not permitted), a status of ”3502 Volume busy“ is returned. If no volume is mounted, or the volume mounted cannot be appended to, a status of ”3503 Volume not mounted“ is returned. If no block with the given volume session ID and the correct client ID number appears in the given first file and block for the volume, a status of ”3505 Session notfound‘‘ is returned.

Read data = <Ticket> > <Block> The specified Block of data from open read session with the specified Ticket number is returned, with a status of 3000 OK followed by a ”Length = *size*“ line giving the length in bytes of the block data which immediately follows. Blocks must be retrieved in ascending order, but blocks may be skipped. If a block number greater than the largest stored on the volume is requested, a status of ”3201 End of volume“ is returned. If a block number greater than the largest in the file is requested, a status of ”3401 End of file‘‘ is returned.

Read close session = <Ticket> The read session with Ticket number is closed. A read session may be closed at any time; you needn’t read all its blocks before closing it.

by January 30th, MM

7.4 SD Data Structures

In the Storage daemon, there is a Device resource (i.e. from conf file) that describes each physical device. When the physical device is used it is controled by the DEVICE structure (defined in dev.h), and typically refered to as dev in the C++ code. Anyone writing or reading a physical device must ultimately get a lock on the DEVICE structure – this controls the device. However, multiple Jobs (defined by a JCR structure src/jcr.h) can be writing a physical DEVICE at the same time (of course they are sequenced by locking the DEVICE structure). There are a lot of job dependent “device” variables that may be different for each Job such as spooling (one job may spool and another may not, and when a job is spooling, it must have an i/o packet open, each job has its own record and block structures, . . .), so there is a device control record or DCR that is the primary way of interfacing to the physical device. The DCR contains all the job specific data as well as a pointer to the Device resource (DEVRES structure) and the physical DEVICE structure.

Now if a job is writing to two devices (it could be writing two separate streams to the same device), it must have two DCRs. Today, the code only permits one. This won’t be hard to change, but it is new code.

Today three jobs (threads), two physical devices each job writes to only one device:

```
Job1 -> DCR1 -> DEVICE1
Job2 -> DCR2 -> DEVICE1
Job3 -> DCR3 -> DEVICE2
```

To be implemented three jobs, three physical devices, but job1 is writing simultaneously to three devices:

```
Job1 -> DCR1 -> DEVICE1
        -> DCR4 -> DEVICE2
        -> DCR5 -> DEVICE3
Job2 -> DCR2 -> DEVICE1
Job3 -> DCR3 -> DEVICE2
```

```
Job = job control record
DCR = Job contorl data for a specific device
DEVICE = Device only control data
```

CHAPTER

EIGHT

CATALOG SERVICES

8.1 General

This chapter is intended to be a technical discussion of the Catalog services and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bareos**.

The **Bareos Catalog** services consist of the programs that provide the SQL database engine for storage and retrieval of all information concerning files that were backed up and their locations on the storage media.

We have investigated the possibility of using the following SQL engines for Bareos: Beagle, mSQL, GNU SQL, PostgreSQL, SQLite, Oracle, and MySQL. Each presents certain problems with either licensing or maturity. At present, we have chosen for development purposes to use MySQL, PostgreSQL and SQLite. MySQL was chosen because it is fast, proven to be reliable, widely used, and actively being developed. MySQL is released under the GNU GPL license. PostgreSQL was chosen because it is a full-featured, very mature database, and because Dan Langille did the Bareos driver for it. PostgreSQL is distributed under the BSD license. SQLite was chosen because it is small, efficient, and can be directly embedded in **Bareos** thus requiring much less effort from the system administrator or person building **Bareos**. In our testing SQLite has performed very well, and for the functions that we use, it has never encountered any errors except that it does not appear to handle databases larger than 2GBytes. That said, we would not recommend it for serious production use.

The Bareos SQL code has been written in a manner that will allow it to be easily modified to support any of the current SQL database systems on the market (for example: mSQL, iODBC, unixODBC, Solid, OpenLink ODBC, EasySoft ODBC, InterBase, Oracle8, Oracle7, and DB2).

If you do not specify either **‘‘–‘‘with-mysql** or **‘‘–‘‘with-postgresql** or **‘‘–‘‘with-sqlite** on the ./configure line, Bareos will use its minimalist internal database. This database is kept for build reasons but is no longer supported. Bareos **requires** one of the three databases (MySQL, PostgreSQL, or SQLite) to run.

8.1.1 Filenames and Maximum Filename Length

In general, either MySQL, PostgreSQL or SQLite permit storing arbitrary long path names and file names in the catalog database. In practice, there still may be one or two places in the Catalog interface code that restrict the maximum path length to 512 characters and the maximum file name length to 512 characters. These restrictions are believed to have been removed. Please note, these restrictions apply only to the Catalog database and thus to your ability to list online the files saved during any job. All information received and stored by the Storage daemon (normally on tape) allows and handles arbitrarily long path and filenames.

8.1.2 Database Table Design

All discussions that follow pertain to the MySQL database. The details for the PostgreSQL and SQLite databases are essentially identical except for that all fields in the SQLite database are stored as ASCII text and some of the database

creation statements are a bit different. The details of the internal Bareos catalog are not discussed here.

Because the Catalog database may contain very large amounts of data for large sites, we have made a modest attempt to normalize the data tables to reduce redundant information. While reducing the size of the database significantly, it does, unfortunately, add some complications to the structures.

In simple terms, the Catalog database must contain a record of all Jobs run by Bareos, and for each Job, it must maintain a list of all files saved, with their File Attributes (permissions, create date, ...), and the location and Media on which the file is stored. This is seemingly a simple task, but it represents a huge amount interlinked data. Note: the list of files and their attributes is not maintained when using the internal Bareos database. The data stored in the File records, which allows the user or administrator to obtain a list of all files backed up during a job, is by far the largest volume of information put into the Catalog database.

Although the Catalog database has been designed to handle backup data for multiple clients, some users may want to maintain multiple databases, one for each machine to be backed up. This reduces the risk of confusion of accidental restoring a file to the wrong machine as well as reducing the amount of data in a single database, thus increasing efficiency and reducing the impact of a lost or damaged database.

8.2 Sequence of Creation of Records for a Save Job

Start with StartDate, ClientName, Filename, Path, Attributes, MediaName, MediaCoordinates. (PartNumber, NumParts). In the steps below, "Create new" means to create a new record whether or not it is unique. "Create unique" means each record in the database should be unique. Thus, one must first search to see if the record exists, and only if not should a new one be created, otherwise the existing RecordId should be used.

1. Create new Job record with StartDate; save JobId
2. Create unique Media record; save MediaId
3. Create unique Client record; save ClientId
4. Create unique Filename record; save FilenameId
5. Create unique Path record; save PathId
6. Create unique Attribute record; save AttributeId store ClientId, FilenameId, PathId, and Attributes
7. Create new File record store JobId, AttributeId, MediaCoordinates, etc
8. Repeat steps 4 through 8 for each file
9. Create a JobMedia record; save MediaId
10. Update Job record filling in EndDate and other Job statistics

8.3 Database Tables

8.3.1 Filename

Column Name	Data Type	Remark
FilenameId	integer	Primary Key
Name	Blob	Filename

The **Filename** table shown above contains the name of each file backed up with the path removed. If different directories or machines contain the same filename, only one copy will be saved in this table.

8.3.2 Path

Column Name	Data Type	Remark
PathId	integer	Primary Key
Path	Blob	Full Path

The **Path** table contains shown above the path or directory names of all directories on the system or systems. The filename and any MSDOS disk name are stripped off. As with the filename, only one copy of each directory name is kept regardless of how many machines or drives have the same directory. These path names should be stored in Unix path name format.

Some simple testing on a Linux file system indicates that separating the filename and the path may be more complication than is warranted by the space savings. For example, this system has a total of 89,097 files, 60,467 of which have unique filenames, and there are 4,374 unique paths.

Finding all those files and doing two stats() per file takes an average wall clock time of 1 min 35 seconds on a 400MHz machine running RedHat 6.1 Linux.

Finding all those files and putting them directly into a MySQL database with the path and filename defined as TEXT, which is variable length up to 65,535 characters takes 19 mins 31 seconds and creates a 27.6 MByte database.

Doing the same thing, but inserting them into Blob fields with the filename indexed on the first 30 characters and the path name indexed on the 255 (max) characters takes 5 mins 18 seconds and creates a 5.24 MB database. Rerunning the job (with the database already created) takes about 2 mins 50 seconds.

Running the same as the last one (Path and Filename Blob), but Filename indexed on the first 30 characters and the Path on the first 50 characters (linear search done there after) takes 5 mins on the average and creates a 3.4 MB database. Rerunning with the data already in the DB takes 3 mins 35 seconds.

Finally, saving only the full path name rather than splitting the path and the file, and indexing it on the first 50 characters takes 6 mins 43 seconds and creates a 7.35 MB database.

8.3.3 File

Column Name	Data Type	Remark
FileId	integer	Primary Key
FileIndex	integer	The sequential file number in the Job
JobId	integer	Link to Job Record
PathId	integer	Link to Path Record
FilenameId	integer	Link to Filename Record
DeltaSeq	smallint	
MarkId	integer	Used to mark files during Verify Jobs
LStat	tinyblob	File attributes in base64 encoding
MD5	tinyblob	MD5/SHA1 signature in base64 encoding

The **File** table shown above contains one entry for each file backed up by Bareos. Thus a file that is backed up multiple times (as is normal) will have multiple entries in the File table. This will probably be the table with the most number of records. Consequently, it is essential to keep the size of this record to an absolute minimum. At the same time, this table must contain all the information (or pointers to the information) about the file and where it is backed up. Since a file may be backed up many times without having changed, the path and filename are stored in separate tables.

This table contains by far the largest amount of information in the Catalog database, both from the stand point of number of records, and the stand point of total database size. As a consequence, the user must take care to periodically reduce the number of File records using the **retention** command in the Console program.

As MD5 hash (also termed message digests) consists of 128-bit (16-byte). A typical format (eg. `md5sum`, ...) to represent them is as a sequence of 32 hexadecimal digits. However, in the **MD5** column, the digest is represented as

base64.

To compare a Bareos digest with command line tools, you have to use

```
openssl dgst -md5 -binary $PATH_OF_YOUR_FILE | base64
```

Also you have to note, that even the table column is named **MD5**, it used to store any kind of digest (MD5, SHA1, ...). This is not directly indicated by the value, however, you can determine the type depending of the length of the digest.

8.3.4 Job / JobHisto

Column Name	Data Type	Remark
JobId	integer	Primary Key
Job	tinyblob	Unique Job Name
Name	tinyblob	Job Name
Type	binary(1)	Job Type: Backup, Copy, Clone, Archive, Migration
Level	binary(1)	Job Level
ClientId	integer	Client index
JobStatu s	binary(1)	Job Termination Status
SchedTim e	datetime	Time/date when Job scheduled
StartTim e	datetime	Time/date when Job started
EndTime	datetime	Time/date when Job ended
ReadEndT ime	datetime	Time/date when original Job ended
JobTDate	bigint	Start day in Unix format but 64 bits; used for Retention period.
VolSessi onId	integer	Unique Volume Session ID
VolSessi onTime	integer	Unique Volume Session Time
JobFiles	integer	Number of files saved in Job
JobBytes	bigint	Number of bytes saved in Job
JobError s	integer	Number of errors during Job
JobMissi ngFiles	integer	Number of files not saved (not yet used)
PoolId	integer	Link to Pool Record
FileSetI d	integer	Link to FileSet Record
PrioJobI d	integer	Link to prior Job Record when migrated
PurgedFi les	tiny integer	Set when all File records purged
HasBase	tiny integer	Set when Base Job run

The **Job** table contains one record for each Job run by Bareos. Thus normally, there will be one per day per machine added to the database. Note, the JobId is used to index Job records in the database, and it often is shown to the user in the Console program. However, care must be taken with its use as it is not unique from database to database. For example, the user may have a database for Client data saved on machine Rufus and another database for Client data saved on machine Roxie. In this case, the two database will each have JobIds that match those in another database. For a unique reference to a Job, see Job below.

The Name field of the Job record corresponds to the Name resource record given in the Director's configuration file. Thus it is a generic name, and it will be normal to find many Jobs (or even all Jobs) with the same Name.

The Job field contains a combination of the Name and the schedule time of the Job by the Director. Thus for a given Director, even with multiple Catalog databases, the Job will contain a unique name that represents the Job.

For a given Storage daemon, the VolSessionId and VolSessionTime form a unique identification of the Job. This will be the case even if multiple Directors are using the same Storage daemon.

The JobStatus field specifies how the job terminated.

The JobHisto table is the same as the Job table, but it keeps long term statistics (i.e. it is not pruned with the Job).

8.3.5 FileSet

Column Name	Data Type	Remark
FileSetId	integer	Primary Key
FileSet	tinyblob	FileSet name
MD5	tinyblob	MD5 checksum of FileSet
CreateTime	datetime	Time and date Fileset created

The **FileSet** table contains one entry for each FileSet that is used. The MD5 signature is kept to ensure that if the user changes anything inside the FileSet, it will be detected and the new FileSet will be used. This is particularly important when doing an incremental update. If the user deletes a file or adds a file, we need to ensure that a Full backup is done prior to the next incremental.

8.3.6 JobMedia

Column Name	Data Type	Remark
JobMed iaId	intege r	Primary Key
JobId	intege r	Link to Job Record
MediaI d	intege r	Link to Media Record
FirstI ndex	intege r	The index (sequence number) of the first file written for this Job to the Media
LastIn dex	intege r	The index of the last file written for this Job to the Media
StartF ile	intege r	Tape: the physical media file mark number of the first block written for this Job.
		Other: upper 32-bit of the position of the first block written for this Job.
EndFil e	intege r	Tape: the physical media file mark number of the last block written for this Job.
		Other: upper 32-bit of the position of the last block written for this Job.
StartB lock	intege r	Tape: the number of the first block written for this Job
		Other: lower 32-bit of the position of the first block written for this Job.
Endblo ck	intege r	Tape: the number of the last block written for this Job
		Other: lower 32-bit of the position of the last block written for this Job.
VolInd ex	intege r	The Volume use sequence number within the Job

The **JobMedia** table contains one entry at the following: start of the job, start of each new tape file mark, start of each new tape, end of the job. You will have 2 or more JobMedia records per Job.

Tape Volume

The number ob records depends on the "Maximum File Size" specified in the Device resource. This record allows Bareos to efficiently position close to any given file in a backup. For restoring a full Job, these records are not very important, but if you want to retrieve a single file that was written near the end of a 100GB backup, the JobMedia records can speed it up by orders of magnitude by permitting forward spacing files and blocks rather than reading the whole 100GB backup.

Other Volume

StartFile and StartBlock are both 32-bit integer values. However, as the position on a disk volume is specified in bytes, we need this to be a 64-bit value.

Therefore, the start position is calculated as:

```
StartPosition = StartFile * 4294967296 + StartBlock
```

The end position of a job on a volume can be determined by:

```
EndPosition = EndFile * 4294967296 + EndBlock
```

Be aware, that you can not assume, that the job size on a volume is `EndPosition -StartPosition`. When interleaving is used other jobs can also be stored between Start- and EndPosition.

```
EndPosition - StartPosition >= JobSizeOnThisMedia
```

8.3.7 Volume (Media)

Column Name	Data Type	Remark
MediaId	integer	Primary Key
VolumeName	tinyblob	Volume name
Slot	integer	Autochanger Slot number or zero
PoolId	integer	Link to Pool Record
MediaType	tinyblob	The MediaType supplied by the user
MediaTypeId	integer	The MediaTypeId
LabelType	tinyint	The type of label on the Volume
FirstWritten	datetime	Time/date when first written
LastWritten	datetime	Time/date when last written
LabelDate	datetime	Time/date when tape labeled
VolJobs	integer	Number of jobs written to this media
VolFiles	integer	Number of files written to this media
VolBlocks	integer	Number of blocks written to this media
VolMounts	integer	Number of time media mounted
VolBytes	bigint	Number of bytes saved in Job
VolParts	integer	The number of parts for a Volume (DVD)
VolErrors	integer	Number of errors during Job
VolWrites	integer	Number of writes to media
MaxVolBytes	bigint	Maximum bytes to put on this media
VolCapacityByte s	bigint	Capacity estimate for this volume
VolStatus	enum	Status of media: Full, Archive, Append, Recycle, Read-Only, Disabled, Error, Busy
Enabled	tinyint	Whether or not Volume can be written
Recycle	tinyint	Whether or not Bareos can recycle the Volumes: Yes, No
ActionOnPurge	tinyint	What happens to a Volume after purging
VolRetention	bigint	64 bit seconds until expiration
VolUseDureation	bigint	64 bit seconds volume can be used
MaxVolJobs	integer	maximum jobs to put on Volume
MaxVolFiles	integer	maximume EOF marks to put on Volume
InChanger	tinyint	Whether or not Volume in autochanger
StorageId	integer	Storage record ID
DeviceId	integer	Device record ID
MediaAddressing	integer	Method of addressing media
VolReadTime	bigint	Time Reading Volume
VolWriteTime	bigint	Time Writing Volume
EndFile	integer	End File number of Volume
EndBlock	integer	End block number of Volume
LocationId	integer	Location record ID
RecycleCount	integer	Number of times recycled
InitialWrite	datetime	When Volume first written

Continued on next page

Table 8.1 – continued from previous page

Column Name	Data Type	Remark
ScratchPoolId	integer	Id of Scratch Pool
RecyclePoolId	integer	Pool ID where to recycle Volume
Comment	blob	User text field

The **Volume** table (internally referred to as the Media table) contains one entry for each volume, that is each tape, cassette (8mm, DLT, DAT, . . .), or file on which information is or was backed up. There is one Volume record created for each of the NumVols specified in the Pool resource record.

8.3.8 Pool

Column Name	Data Type	Remark
PoolId	integer	Primary Key
Name	Tinyblob	Pool Name
NumVols	Integer	Number of Volumes in the Pool
MaxVols	Integer	Maximum Volumes in the Pool
UseOnce	tinyint	Use volume once
UseCatalog	tinyint	Set to use catalog
AcceptAnyVolume	tinyint	Accept any volume from Pool
VolRetention	bigint	64 bit seconds to retain volume
VolUseDuration	bigint	64 bit seconds volume can be used
MaxVolJobs	integer	max jobs on volume
MaxVolFiles	integer	max EOF marks to put on Volume
MaxVolBytes	bigint	max bytes to write on Volume
AutoPrune	tinyint	yes or no for autopruning
Recycle	tinyint	yes or no for allowing auto recycling of Volume
ActionOnPurge	tinyint	Default Volume ActionOnPurge
PoolType	enum	Backup, Copy, Cloned, Archive, Migration
LabelType	tinyint	Type of label ANSI/Bareos
LabelFormat	Tinyblob	Label format
Enabled	tinyint	Whether or not Volume can be written
ScratchPoolId	integer	Id of Scratch Pool
RecyclePoolId	integer	Pool ID where to recycle Volume
NextPoolId	integer	Pool ID of next Pool
MigrationHighBytes	bigint	High water mark for migration
MigrationLowBytes	bigint	Low water mark for migration
MigrationTime	bigint	Time before migration

The **Pool** table contains one entry for each media pool controlled by Bareos in this database. One media record exists for each of the NumVols contained in the Pool. The PoolType is a Bareos defined keyword. The MediaType is defined by the administrator, and corresponds to the MediaType specified in the Director's Storage definition record. The CurrentVol is the sequence number of the Media record for the current volume.

8.3.9 Client

Column Name	Data Type	Remark
ClientId	integer	Primary Key
Name	TinyBlob	File Services Name
UName	TinyBlob	uname -a from Client (not yet used)
AutoPrune	tinyint	yes or no for autopruning
FileRetention	bigint	64 bit seconds to retain Files
JobRentention	bigint	64 bit seconds to retain Job

The **Client** table contains one entry for each machine backed up by Bareos in this database. Normally the Name is a fully qualified domain name.

8.3.10 Storage

Column Name	Data Type	Remark
StorageId	integer	Unique Id
Name	tinyblob	Resource name of Storage device
AutoChanger	tinyint	Set if it is an autochanger

The **Storage** table contains one entry for each Storage used.

8.3.11 Counter

Column Name	Data Type	Remark
Counter	tinyblob	Counter name
MinValue	integer	Start/Min value for counter
MaxValue	integer	Max value for counter
CurrentValue	integer	Current counter value
WrapCounter	tinyblob	Name of another counter

The **Counter** table contains one entry for each permanent counter defined by the user.

8.3.12 Log

Column Name	Data Type	Remark
LogIdId	integer	Primary Key
JobId	integer	Points to Job record
Time	datetime	Time/date log record created
LogText	blob	Log text

The **Log** table contains a log of all Job output.

8.3.13 Location

Column Name	Data Type	Remark
LocationId	integer	Primary Key
Location	tinyblob	Text defining location
Cost	integer	Relative cost of obtaining Volume
Enabled	tinyint	Whether or not Volume is enabled

The **Location** table defines where a Volume is physically.

8.3.14 LocationLog

Column Name	Data Type	Remark
LocLogId	integer	Primary Key
Date	datetime	Time/date log record created
MediaId	integer	Points to Media record
LocationI d	integer	Points to Location record
NewVolSta tus	integer	enum: Full, Archive, Append, Recycle, Purged Read-only, Disabled, Error, Busy, Used, Cleaning
Enabled	tinyint	Whether or not Volume is enabled

The **LocationLog** table contains a log of all Job output.

8.3.15 Version

Column Name	Data Type	Remark
VersionId	integer	Primary Key

The **Version** table defines the Bareos database version number. Bareos checks this number before reading the database to ensure that it is compatible with the Bareos binary file.

8.3.16 BaseFiles

Column Name	Data Type	Remark
BaseId	integer	Primary Key
BaseJobId	integer	JobId of Base Job
JobId	integer	Reference to Job
FileId	integer	Reference to File
FileIndex	integer	File Index number

The **BaseFiles** table contains all the File references for a particular JobId that point to a Base file – i.e. they were previously saved and hence were not saved in the current JobId but in BaseJobId under FileId. FileIndex is the index of the file, and is used for optimization of Restore jobs to prevent the need to read the FileId record when creating the in memory tree. This record is not yet implemented.

CHAPTER

NINE

STORAGE MEDIA OUTPUT FORMAT

9.1 General

This document describes the media format written by the Storage daemon. The Storage daemon reads and writes in units of blocks. Blocks contain records. Each block has a block header followed by records, and each record has a record header followed by record data.

This chapter is intended to be a technical discussion of the Media Format.

9.2 Definitions

Block A block represents the primitive unit of information that the Storage daemon reads and writes to a physical device. Normally, for a tape device, it will be the same as a tape block. The Storage daemon always reads and writes blocks. A block consists of *block header* information followed by records. Clients of the Storage daemon (the File daemon) normally never see blocks. However, some of the Storage tools (*bls*, bscan, bextract, ...) may be use block header information. From Bacula >= 1.27 and therefore Bareos, each block contains only records of a single job.

Record A record consists of a Record Header, which is managed by the Storage daemon and Record Data, which is the data received from the Client. A record is the primitive unit of information sent to and from the Storage daemon by the Client (File daemon) programs. The details are described below.

JobId A number assigned by the Director daemon for a particular job. This number will be unique for that particular Director (Catalog). The daemons use this number to keep track of individual jobs. Within the Storage daemon, the JobId may not be unique if several Directors are accessing the Storage daemon simultaneously.

Session A Session is a concept used in the Storage daemon corresponds one to one to a Job with the exception that each session is uniquely identified within the Storage daemon by a unique *VolSessionId*/*VolSessionTime* pair.

VolSessionId A unique sequential number assigned by the Storage daemon to a particular session (Job) it is having with a File daemon. This number is sequential since the start of the daemon. This number by itself is not unique to the given Volume, but with the *VolSessionTime*, it is unique.

VolSessionTime A unique number assigned by the Storage daemon to a particular Storage daemon execution. It is actually the Unix `time_t` value of when the Storage daemon began execution cast to a 32 bit unsigned integer. The combination of the *VolSessionId* and the VolSessionTime for a given Storage daemon is guaranteed to be unique for each Job (or session).

File Index A sequential number beginning at one assigned by the File daemon to the files within a job that are sent to the Storage daemon for backup. The Storage daemon ensures that this number is greater than zero and sequential. Note, the Storage daemon uses negative FileIndexes to flag *Session Start* and *End* labels as well as End of *Volume Labels*. Thus, the combination of *VolSessionId*, *VolSessionTime* and FileIndex uniquely identifies the records for a single file written to a Volume.

Stream While writing the information for any particular file to the Volume, there can be any number of distinct pieces of information about that file, e.g. the attributes, the file data, ... The Stream indicates what piece of data it is, and it is an arbitrary number assigned by the File daemon to the parts (Unix attributes, Win32 attributes, data, compressed data, ...) of a file that are sent to the Storage daemon. The Storage daemon has no knowledge of the details of a Stream; it simply represents a numbered stream of bytes. The data for a given stream may be passed to the Storage daemon in a single or multiple *records*.

Block Header A block header consists of a block identification ("BB02"), a block length in bytes (typically 64,512) a checksum, and sequential block number. Each block starts with a Block Header and is followed by Records. Current block headers also contain the *VolSessionId* and *VolSessionTime* for the records written to that block.

Record Header A record header contains the *VolSessionId*, the *VolSessionTime*, the *FileIndex*, the *Stream* type and the size of the *data record* which follows. The Record Header is always immediately followed by a *Data Record* if the size given in the Header is greater than zero.

Note, for Block headers of level BB02 (Bacula >= 1.27 and Bareos), the Record header as written to tape does not contain the Volume Session Id and the Volume Session Time as these two fields are stored in the BB02 *Block Header*. The in-memory record header does have those fields for convenience.

Data Record A data record consists of a binary *Stream* of bytes and is always preceded by a *Record Header*. The details of the meaning of the binary stream of bytes are unknown to the Storage daemon, but the Client programs (File daemon) defines and thus knows the details of each record type.

Label *Volume*, *Start Of Session* and *End Of Session* are special *records* that are used as Labels.

Volume Label A label placed by the Storage daemon at the beginning of each storage volume. It contains general information about the volume. It is written in *record* format. The Storage daemon manages Volume Labels, and if the client wants, he may also read them.

Start Of Session Record The Start Of Session (SOS) record is a special *record* placed by the Storage daemon on the storage medium as the first record of an append session job with a File daemon. This record is useful for finding the beginning of a particular session (Job), since no records with the same *VolSessionId* and *VolSessionTime* will precede this record. This record is not normally visible outside of the Storage daemon. The Begin Session Label is similar to the *Volume Label* except that it contains additional information pertaining to the *Session*.

End Of Session Record The End Of Session (EOS) Record is a special *record* placed by the Storage daemon on the storage medium as the last record of an append session job with a File daemon. The End Session Record is distinguished by a *FileIndex* with a value of minus two (-2). This record is useful for detecting the end of a particular session since no records with the same *VolSessionId* and *VolSessionTime* will follow this record. This record is not normally visible outside of the Storage daemon. The End Session Label is similar to the *Volume Label* except that it contains additional information pertaining to the *Session*.

9.3 Overall Format

A Bareos output file consists of Blocks of data. Each block contains a block header followed by records. Each record consists of a record header followed by the record data. The first record on a tape will always be the Volume Label Record.

No Record Header will be split across Bareos blocks. However, Record Data may be split across any number of Bareos blocks. Obviously this will not be the case for the Volume Label which will always be smaller than the Bareos Block size.

To simplify reading tapes, the Start of Session (SOS) and End of Session (EOS) records are never split across blocks. If this is about to happen, Bareos will write a short block before writing the session record (actually, the SOS record should always be the first record in a block, excepting perhaps the Volume label).

Due to hardware limitations, the last block written to the tape may not be fully written. If your drive permits backspace record, Bareos will backup over the last record written on the tape, re-read it and verify that it was correctly written.

When a new tape is mounted Bareos will write the full contents of the partially written block to the new tape ensuring that there is no loss of data. When reading a tape, Bareos will discard any block that is not totally written, thus ensuring that there is no duplication of data. In addition, since Bareos blocks are sequentially numbered within a Job, it is easy to ensure that no block is missing or duplicated.

9.3.1 Storage Daemon File Output Format

The file storage and tape storage formats are identical except that tape records are by default blocked into blocks of 64,512 bytes, except for the last block, which is the actual number of bytes written rounded up to a multiple of 1024 whereas the last record of file storage is not rounded up. Each Session written to tape is terminated with an End of File mark (this will be removed later). Sessions written to file are simply appended to the end of the file.

9.4 Serialization

All *Block Headers*, *Record Headers* and *Label Records* are written using Bareos's serialization routines. These routines guarantee that the data is written to the output volume in a machine independent format.

9.5 Block Header

The current Block Header version is **BB02**. (The prior version *BB01* is deprecated since Bacula 1.27.)

Each session or Job use their own private blocks.

The format of a *Block Header* is:

```
uint32_t CheckSum;                /* Block check sum */
uint32_t BlockSize;               /* Block byte size including the header */
uint32_t BlockNumber;             /* Block number */
char ID[4] = "BB02";              /* Identification and block level */
uint32_t VolSessionId;            /* Session Id for Job */
uint32_t VolSessionTime;          /* Session Time for Job */
```

The Block Header is a fixed length and fixed format.

The CheckSum field is a 32 bit checksum of the block data and the block header but not including the CheckSum field.

The Block Header is always immediately followed by a *Record Header*. If the tape is damaged, a Bareos utility will be able to recover as much information as possible from the tape by recovering blocks which are valid. The Block header is written using the Bareos serialization routines and thus is guaranteed to be in machine independent format.

9.6 Record Header

Each binary data record is preceded by a Record Header. The Record Header is fixed length and fixed format, whereas the binary data record is of variable length. The Record Header is written using the Bareos serialization routines and thus is guaranteed to be in machine independent format.

The format of the Record Header is:

```
int32_t FileIndex;    /* File index supplied by File daemon */
int32_t Stream;       /* Stream number supplied by File daemon */
uint32_t DataSize;    /* size of following data record in bytes */
```

This version 2 Record Header is written to the medium when using Version BB02 *Block Headers*.

This record is followed by the binary Stream data of DataSize bytes, followed by another Record Header record and the binary stream data. For the definitive definition of this record, see record.h in the src/stored directory.

Additional notes on the above:

FileIndex is a sequential file number within a job. The Storage daemon requires this index to be greater than zero and sequential. Note, however, that the File daemon may send multiple Streams for the same FileIndex. In addition, the Storage daemon uses negative FileIndices to hold the Begin Session Label, the End Session Label, and the End of Volume Label.

Stream is defined by the File daemon and is used to identify separate parts of the data saved for each file (Unix attributes, Win32 attributes, file data, compressed file data, sparse file data, ...). The Storage Daemon has no idea of what a Stream is or what it contains except that the Stream is required to be a positive integer. Negative Stream numbers are used internally by the Storage daemon to indicate that the record is a continuation of the previous record (the previous record would not entirely fit in the block).

For Start Session and End Session Labels (where the FileIndex is negative), the Storage daemon uses the Stream field to contain the JobId.

The current stream definitions are:

```
#define STREAM_UNIX_ATTRIBUTES     1    /* Generic Unix attributes */
#define STREAM_FILE_DATA           2    /* Standard uncompressed data */
#define STREAM_MD5_SIGNATURE       3    /* MD5 signature for the file */
#define STREAM_GZIP_DATA           4    /* GZip compressed file data */
/* Extended Unix attributes with Win32 Extended data.  Deprecated. */
#define STREAM_UNIX_ATTRIBUTES_EX 5    /* Extended Unix attr for Win32 EX */
#define STREAM_SPARSE_DATA         6    /* Sparse data stream */
#define STREAM_SPARSE_GZIP_DATA    7
#define STREAM_PROGRAM_NAMES       8    /* program names for program data */
#define STREAM_PROGRAM_DATA        9    /* Data needing program */
#define STREAM_SHA1_SIGNATURE     10    /* SHA1 signature for the file */
#define STREAM_WIN32_DATA         11    /* Win32 BackupRead data */
#define STREAM_WIN32_GZIP_DATA    12    /* Gzipped Win32 BackupRead data */
#define STREAM_MACOS_FORK_DATA    13    /* Mac resource fork */
#define STREAM_HFSPLUS_ATTRIBUTES 14    /* Mac OS extra attributes */
#define STREAM_UNIX_ATTRIBUTES_ACCESS_ACL 15 /* Standard ACL attributes on UNIX */
#define STREAM_UNIX_ATTRIBUTES_DEFAULT_ACL 16 /* Default ACL attributes on UNIX */
```

DataSize is the size in bytes of the binary data record that follows the Session Record header. The Storage Daemon has no idea of the actual contents of the binary data record. For standard Unix files, the data record typically contains the file attributes or the file data. For a sparse file the first 64 bits of the file data contains the storage address for the data block.

The Record Header is never split across two blocks. If there is not enough room in a block for the full Record Header, the block is padded to the end with zeros and the Record Header begins in the next block. The data record, on the other hand, may be split across multiple blocks and even multiple physical volumes. When a data record is split, the second (and possibly subsequent) piece of the data is preceded by a new Record Header. Thus each piece of data is always immediately preceded by a Record Header. When reading a record, if Bareos finds only part of the data in the first record, it will automatically read the next record and concatenate the data record to form a full data record.

9.7 Volume Label Format

Tape volume labels are created by the Storage daemon in response to a **label** command given to the Console program, or alternatively by the **btape** program. created. Each volume is labeled with the following information using the

Bareos serialization routines, which guarantee machine byte order independence.

For Bacula versions 1.27 and later, the Volume Label Format is:

```
char Id[32];              /* Bacula 1.0 Immortal\n */
uint32_t VerNum;          /* Label version number */
/* VerNum 11 and greater Bacula 1.27 and later */
btime_t   label_btime;    /* Time/date tape labeled */
btime_t   write_btime;    /* Time/date tape first written */
/* The following are 0 in VerNum 11 and greater */
float64_t write_date;     /* Date this label written */
float64_t write_time;     /* Time this label written */
char VolName[128];        /* Volume name */
char PrevVolName[128];    /* Previous Volume Name */
char PoolName[128];       /* Pool name */
char PoolType[128];       /* Pool type */
char MediaType[128];      /* Type of this media */
char HostName[128];       /* Host name of writing computer */
char LabelProg[32];       /* Label program name */
char ProgVersion[32];     /* Program version */
char ProgDate[32];        /* Program build date/time */
```

Note, the LabelType (Volume Label, Volume PreLabel, Session Start Label, ...) is stored in the record FileIndex field of the Record Header and does not appear in the data part of the record.

9.8 Session Label

The Session Label is written at the beginning and end of each session as well as the last record on the physical medium. It has the following binary format:

```
char Id[32];              /* Bacula/Bareos Immortal ... */
uint32_t VerNum;          /* Label version number */
uint32_t JobId;           /* Job id */
uint32_t VolumeIndex;     /* sequence no of vol */
/* Prior to VerNum 11 */
float64_t write_date;     /* Date this label written */
/* VerNum 11 and greater */
btime_t   write_btime;    /* time/date record written */
/* The following is zero VerNum 11 and greater */
float64_t write_time;    /* Time this label written */
char PoolName[128];        /* Pool name */
char PoolType[128];        /* Pool type */
char JobName[128];         /* base Job name */
char ClientName[128];
/* Added in VerNum 10 */
char Job[128];             /* Unique Job name */
char FileSetName[128];     /* FileSet name */
uint32_t JobType;
uint32_t JobLevel;
```

In addition, the *EOS* label contains:

```
/* The remainder are part of EOS label only */
uint32_t JobFiles;
uint64_t JobBytes;
uint32_t start_block;
uint32_t end_block;
```

```
uint32_t start_file;
uint32_t end_file;
uint32_t JobErrors;
uint32_t JobStatus          /* Job termination code, since VerNum >= 11 */
```

Note, the LabelType (Volume Label, Volume PreLabel, Session Start Label, ...) is stored in the record FileIndex field and does not appear in the data part of the record. Also, the Stream field of the Record Header contains the JobId. This permits quick filtering without actually reading all the session data in many cases.

9.9 Overall Storage Format

```
            Bacula/Bareos Tape Format
                  6 June 2001
        Version BB02 added 28 September 2002
        Version BB01 is the old deprecated format.
A Bareos tape is composed of tape Blocks.  Each block
  has a Block header followed by the block data. Block
  Data consists of Records. Records consist of Record
  Headers followed by Record Data.
:=======================================================:
|                                                       |
|                 Block Header (24 bytes)               |
|                                                       |
|-------------------------------------------------------|
|                                                       |
|               Record Header (12 bytes)                |
|                                                       |
|-------------------------------------------------------|
|                                                       |
|                  Record Data                          |
|                                                       |
|-------------------------------------------------------|
|                                                       |
|               Record Header (12 bytes)                |
|                                                       |
|-------------------------------------------------------|
|                                                       |
|                        ...                            |
Block Header: the first item in each block. The format is
  shown below.
Partial Data block: occurs if the data from a previous
  block spills over to this block (the normal case except
  for the first block on a tape). However, this partial
      data block is always preceded by a record header.
Record Header: identifies the FileIndex, the Stream
  and the following Record Data size. See below for format.
Record data: arbitrary binary data.
                Block Header Format BB02
:=======================================================:
|               CheckSum         (uint32_t)             |
|-------------------------------------------------------|
|               BlockSize        (uint32_t)             |
|-------------------------------------------------------|
|               BlockNumber      (uint32_t)             |
|-------------------------------------------------------|
```

```
|              "BB02"           (char [4])              |
|-------------------------------------------------------|
|              VolSessionId     (uint32_t)              |
|-------------------------------------------------------|
|              VolSessionTime   (uint32_t)              |
:=======================================================:
BBO2: Serves to identify the block as a
  Bacula/Bareos block and also servers as a block format identifier
  should we ever need to change the format.
BlockSize: is the size in bytes of the block. When reading
  back a block, if the BlockSize does not agree with the
  actual size read, Bareos discards the block.
CheckSum: a checksum for the Block.
BlockNumber: is the sequential block number on the tape.
VolSessionId: a unique sequential number that is assigned
              by the Storage Daemon to a particular Job.
              This number is sequential since the start
              of execution of the daemon.
VolSessionTime: the time/date that the current execution
              of the Storage Daemon started.  It assures
              that the combination of VolSessionId and
              VolSessionTime is unique for all jobs
              written to the tape, even if there was a
              machine crash between two writes.
               Record Header Format BB02
:=======================================================:
|              FileIndex        (int32_t)               |
|-------------------------------------------------------|
|              Stream           (int32_t)               |
|-------------------------------------------------------|
|              DataSize         (uint32_t)              |
:=======================================================:
FileIndex: a sequential file number within a job.  The
              Storage daemon enforces this index to be
              greater than zero and sequential.  Note,
              however, that the File daemon may send
              multiple Streams for the same FileIndex.
              The Storage Daemon uses negative FileIndices
              to identify Session Start and End labels
              as well as the End of Volume labels.
Stream: defined by the File daemon and is intended to be
              used to identify separate parts of the data
              saved for each file (attributes, file data,
              ...).  The Storage Daemon has no idea of
              what a Stream is or what it contains.
DataSize: the size in bytes of the binary data record
              that follows the Session Record header.
              The Storage Daemon has no idea of the
              actual contents of the binary data record.
              For standard Unix files, the data record
              typically contains the file attributes or
              the file data.  For a sparse file
              the first 64 bits of the data contains
              the storage address for the data block.
                    Volume Label
:=======================================================:
|              Id               (32 bytes)              |
|-------------------------------------------------------|
```

```
|               VerNum           (uint32_t)               |
|---------------------------------------------------------|
|               label_date       (float64_t)              |
|               label_btime      (btime_t VerNum 11       |
|---------------------------------------------------------|
|               label_time       (float64_t)              |
|               write_btime      (btime_t VerNum 11       |
|---------------------------------------------------------|
|               write_date       (float64_t)              |
|                   0            (float64_t) VerNum 11    |
|---------------------------------------------------------|
|               write_time       (float64_t)              |
|                   0            (float64_t) VerNum 11    |
|---------------------------------------------------------|
|               VolName          (128 bytes)              |
|---------------------------------------------------------|
|               PrevVolName      (128 bytes)              |
|---------------------------------------------------------|
|               PoolName         (128 bytes)              |
|---------------------------------------------------------|
|               PoolType         (128 bytes)              |
|---------------------------------------------------------|
|               MediaType        (128 bytes)              |
|---------------------------------------------------------|
|               HostName         (128 bytes)              |
|---------------------------------------------------------|
|               LabelProg        (32 bytes)               |
|---------------------------------------------------------|
|               ProgVersion      (32 bytes)               |
|---------------------------------------------------------|
|               ProgDate         (32 bytes)               |
|---------------------------------------------------------|
:=========================================================:

Id: 32 byte identifier "Bacula 1.0 immortal\n"
(old version also recognized:)
Id: 32 byte identifier "Bacula 0.9 mortal\n"
LabelType (Saved in the FileIndex of the Header record).
    PRE_LABEL -1    Volume label on unwritten tape
    VOL_LABEL -2    Volume label after tape written
    EOM_LABEL -3    Label at EOM (not currently implemented)
    SOS_LABEL -4    Start of Session label (format given below)
    EOS_LABEL -5    End of Session label (format given below)
VerNum: 11
label_date: Julian day tape labeled
label_time: Julian time tape labeled
write_date: Julian date tape first used (data written)
write_time: Julian time tape first used (data written)
VolName: "Physical" Volume name
PrevVolName: The VolName of the previous tape (if this tape is
             a continuation of the previous one).
PoolName: Pool Name
PoolType: Pool Type
MediaType: Media Type
HostName: Name of host that is first writing the tape
LabelProg: Name of the program that labeled the tape
ProgVersion: Version of the label program
ProgDate: Date Label program built
```

```
                    Session Label
:=======================================================:
|               Id                (32 bytes)            |
|-------------------------------------------------------|
|               VerNum            (uint32_t)           |
|-------------------------------------------------------|
|               JobId             (uint32_t)           |
|-------------------------------------------------------|
|               write_btime       (btime_t)   VerNum 11  |
|-------------------------------------------------------|
|                  0              (float64_t) VerNum 11  |
|-------------------------------------------------------|
|               PoolName          (128 bytes)          |
|-------------------------------------------------------|
|               PoolType          (128 bytes)          |
|-------------------------------------------------------|
|               JobName           (128 bytes)          |
|-------------------------------------------------------|
|               ClientName        (128 bytes)          |
|-------------------------------------------------------|
|               Job               (128 bytes)          |
|-------------------------------------------------------|
|               FileSetName       (128 bytes)          |
|-------------------------------------------------------|
|               JobType           (uint32_t)           |
|-------------------------------------------------------|
|               JobLevel          (uint32_t)           |
|-------------------------------------------------------|
|               FileSetMD5        (50 bytes)   VerNum 11  |
|-------------------------------------------------------|
        Additional fields in End Of Session Label
|-------------------------------------------------------|
|               JobFiles          (uint32_t)           |
|-------------------------------------------------------|
|               JobBytes          (uint32_t)           |
|-------------------------------------------------------|
|               start_block       (uint32_t)           |
|-------------------------------------------------------|
|               end_block         (uint32_t)           |
|-------------------------------------------------------|
|               start_file        (uint32_t)           |
|-------------------------------------------------------|
|               end_file          (uint32_t)           |
|-------------------------------------------------------|
|               JobErrors         (uint32_t)           |
|-------------------------------------------------------|
|               JobStatus         (uint32_t) VerNum 11   |
:=======================================================:
* => fields deprecated
Id: 32 byte identifier "Bacula 1.0 immortal\n"
LabelType (in FileIndex field of Header):
    EOM_LABEL -3     Label at EOM
    SOS_LABEL -4     Start of Session label
    EOS_LABEL -5     End of Session label
VerNum: 11
JobId: JobId
write_btime: Bareos time/date this tape record written
write_date: Julian date tape this record written - deprecated
```

```
write_time: Julian time tape this record written - deprecated.
PoolName: Pool Name
PoolType: Pool Type
MediaType: Media Type
ClientName: Name of File daemon or Client writing this session
            Not used for EOM_LABEL.
```

9.10 Examine Volumes

9.10.1 bls command

To get these information from actual volumes (disk or tape volumes), the bls command can be used.

- `bls <StorageName> -V <VolumeName>`
  - shows general volume information, jobs and files in these jobs
- `bls <StorageName> -V <VolumeName> -v`
  - shows general volume, block and detailed record information. As files are stored in record, also all files are listed, together with information about sparse, compression, encryption, ...
- `bls <StorageName> -V <VolumeName> -k -vv`
  - shows block and record information. Opposite to the commands before, it also shows all parts of records splitted by block boundaries.

9.11 Unix File Attributes

The Unix File Attributes packet consists of the following:

```
FileIndex Type Filename@FileAttributes@Link @ExtendedAttributes@
```

where

@ represents a byte containing a binary zero.

FileIndex is the sequential file index starting from one assigned by the File daemon.

Type is one of the following:

```
#define FT_LNKSAVED   1    /* hard link to file already saved */
#define FT_REGE       2    /* Regular file but empty */
#define FT_REG        3    /* Regular file */
#define FT_LNK        4    /* Soft Link */
#define FT_DIR        5    /* Directory */
#define FT_SPEC       6    /* Special file -- chr, blk, fifo, sock */
#define FT_NOACCESS   7    /* Not able to access */
#define FT_NOFOLLOW   8    /* Could not follow link */
#define FT_NOSTAT     9    /* Could not stat file */
#define FT_NOCHG     10    /* Incremental option, file not changed */
#define FT_DIRNOCHG  11    /* Incremental option, directory not changed */
#define FT_ISARCH    12    /* Trying to save archive file */
#define FT_NORECURSE 13    /* No recursion into directory */
#define FT_NOFSCHG   14    /* Different file system, prohibited */
#define FT_NOOPEN    15    /* Could not open directory */
```

```
#define FT_RAW       16    /* Raw block device */
#define FT_FIFO      17    /* Raw fifo device */
```

Filename is the fully qualified filename.

FileAttributes consists of the 13 fields of the stat() buffer in ASCII base64 format separated by spaces. These fields and their meanings are shown below. This stat() packet is in Unix format, and MUST be provided (constructed) for ALL systems.

Link when the FT code is FT_LNK or FT_LNKSAVED, the item in question is a Unix link, and this field contains the fully qualified link name. When the FT code is not FT_LNK or FT_LNKSAVED, this field is null.

ExtendedAttributes The exact format of this field is operating system dependent. It contains additional or extended attributes of a system dependent nature. Currently, this field is used only on WIN32 systems where it contains a ASCII base64 representation of the WIN32_FILE_ATTRIBUTE_DATA structure as defined by Windows. The fields in the base64 representation of this structure are like the File-Attributes separated by spaces.

The File-attributes consist of the following:

Stat Name	Unix	Windows	MacOS
st_de v	Device number of filesystem	Drive number	vRefNum
st_in o	Inode number	Always 0	fileID/dirID
st_mo de	File mode	File mode	777 dirs/apps; 666 docs; 444 locked docs
st_nl ink	Number of links to the file	Number of link (only on NTFS)	Always 1
st_ui d	Owner ID	Always 0	Always 0
st_gi d	Group ID	Always 0	Always 0
st_rd ev	Device ID for special files	Drive No.	Always 0
st_si ze	File size in bytes	File size in bytes	Data fork file size in bytes
st_bl ksize	Preferred block size	Always 0	Preferred block size
st_bl ocks	Number of blocks allocated	Always 0	Number of blocks allocated
st_at ime	Last access time since epoch	Last access time since epoch	Last access time -66 years
st_mt ime	Last modify time since epoch	Last modify time since epoch	Last access time -66 years
st_ct ime	Inode change time since epoch	File create time since epoch	File create time -66 years

9.12 Old Deprecated Tape Format

In Bacula <= 1.26, a *Block* could contain *records* from multiple jobs. However, all blocks currently written by Bacula/Bareos are block level BB02, and a given block contains records for only a single job. Different jobs simply have their own private blocks that are intermingled with the other blocks from other jobs on the Volume (previously the records were intermingled within the blocks). Having only records from a single job in any given block permitted moving the VolumeSessionId and VolumeSessionTime (see below) from each record heading to the Block header. This has two advantages: 1. a block can be quickly rejected based on the contents of the header without reading all the records. 2. because there is on the average more than one record per block, less data is written to the Volume for each job.

The format of the Block Header (Bacula <= 1.26) is:

```
uint32_t CheckSum;       /* Block check sum */
uint32_t BlockSize;      /* Block byte size including the header */
uint32_t BlockNumber;    /* Block number */
char ID[4] = "BB01";     /* Identification and block level */
```

The format of the Record Header (Bacula <= 1.26) is:

```
uint32_t VolSessionId;     /* Unique ID for this session */
uint32_t VolSessionTime;   /* Start time/date of session */
int32_t FileIndex;         /* File index supplied by File daemon */
int32_t Stream;            /* Stream number supplied by File daemon */
uint32_t DataSize;         /* size of following data record in bytes */
```

CHAPTER

TEN

BAREOS PORTING NOTES

This document is intended mostly for developers who wish to port Bareos to a system that is not **officially** supported.

It is hoped that Bareos clients will eventually run on every imaginable system that needs backing up. It is also hoped that the Bareos Directory and Storage daemons will run on every system capable of supporting them.

10.1 Porting Requirements

In General, the following holds true:

- Your compiler must provide support for 64 bit signed and unsigned integers.
- You will need a recent copy of the **autoconf** tools loaded on your system (version 2.13 or later). The **autoconf** tools are used to build the configuration program, but are not part of the Bareos source distribution.
- **Bareos** requires a good implementation of pthreads to work.
- The source code has been written with portability in mind and is mostly POSIX compatible. Thus porting to any POSIX compatible operating system should be relatively easy.

10.2 Steps to Take for Porting

- The first step is to ensure that you have version 2.13 or later of the **autoconf** tools loaded. You can skip this step, but making changes to the configuration program will be difficult or impossible.
- The run a **./configure** command in the main source directory and examine the output. It should look something like the following:

```
Configuration on Mon Oct 28 11:42:27 CET 2002:
  Host:                           i686-pc-linux-gnu -- redhat 7.3
  Bareos version:                 1.27 (26 October 2002)
  Source code location:           .
  Install binaries:               /sbin
  Install config files:           /etc/bareos
  C Compiler:                     gcc
  C++ Compiler:                   c++
  Compiler flags:                 -g -O2
  Linker flags:
  Libraries:                      -lpthread
  Statically Linked Tools:        no
  Database found:                 no
  Database type:                  Internal
  Database lib:
```

```
Job Output Email:             root@localhost
Traceback Email:              root@localhost
SMTP Host Address:            localhost
Director Port                 9101
File daemon Port              9102
Storage daemon Port           9103
Working directory             /etc/bareos/working
SQL binaries Directory
Large file support:           yes
readline support:             yes
TCP Wrappers support:         no
ZLIB support:                 yes
enable-smartalloc:            yes
enable-gnome:                 no
gmp support:                  yes
```

The details depend on your system. The first thing to check is that it properly identified your host on the **Host:** line. The first part (added in version 1.27) is the GNU four part identification of your system. The part after the – is your system and the system version. Generally, if your system is not yet supported, you must correct these.

- If the **./configure** does not function properly, you must determine the cause and fix it. Generally, it will be because some required system routine is not available on your machine.
- To correct problems with detection of your system type or with routines and libraries, you must edit the file **<bareos-src>/autoconf/configure.in**. This is the "source" from which **configure** is built. In general, most of the changes for your system will be made in **autoconf/aclocal.m4** in the routine **BA_CHECK_OPSYS** or in the routine **BA_CHECK_OPSYS_DISTNAME**. I have already added the necessary code for most systems, but if yours shows up as **unknown** you will need to make changes. Then as mentioned above, you will need to set a number of system dependent items in **configure.in** in the **case** statement at approximately line 1050 (depending on the Bareos release).
- The items to in the case statement that corresponds to your system are the following:
  - DISTVER – set to the version of your operating system. Typically some form of **uname** obtains it.
  - TAPEDRIVE – the default tape drive. Not too important as the user can set it as an option.
  - PSCMD – set to the **ps** command that will provide the PID in the first field and the program name in the second field. If this is not set properly, the **bareos stop** script will most likely not be able to stop Bareos in all cases.
  - hostname – command to return the base host name (non-qualified) of your system. This is generally the machine name. Not too important as the user can correct this in his configuration file.
  - CFLAGS – set any special compiler flags needed. Many systems need a special flag to make pthreads work. See cygwin for an example.
  - LDFLAGS – set any special loader flags. See cygwin for an example.
  - PTHREAD_LIB – set for any special pthreads flags needed during linking. See freebsd as an example.
  - lld – set so that a "long long int" will be properly edited in a printf() call.
  - llu – set so that a "long long unsigned" will be properly edited in a printf() call.
  - PFILES – set to add any files that you may define is your platform subdirectory. These files are used for installation of automatic system startup of Bareos daemons.
- To rebuild a new version of **configure** from a changed **autoconf/configure.in** you enter **make configure** in the top level Bareos source directory. You must have done a ./configure prior to trying to rebuild the configure script or it will get into an infinite loop.

- If the **make configure** gets into an infinite loop, ctl-c it, then do **./configure** (no options are necessary) and retry the **make configure**, which should now work.
- To rebuild **configure** you will need to have **autoconf** version 2.57-3 or higher loaded. Older versions of autoconf will complain about unknown or bad options, and won't work.
- After you have a working **configure** script, you may need to make a few system dependent changes to the way Bareos works. Generally, these are done in **src/baconfig.h**. You can find a few examples of system dependent changes toward the end of this file. For example, on Irix systems, there is no definition for **socklen_t**, so it is made in this file. If your system has structure alignment requirements, check the definition of BALIGN in this file. Currently, all Bareos allocated memory is aligned on a **double** boundary.
- If you are having problems with Bareos's type definitions, you might look at **src/bc_types.h** where all the types such as **uint32_t**, **uint64_t**, etc. that Bareos uses are defined.

CHAPTER

ELEVEN

API

11.1 General

This document is intended mostly for developers who wish to develop a new GUI interface to **Bareos**.

11.1.1 Minimal Code in Console Program

All the Catalog code is in the Directory (with the exception of `dbcheck` and `bscan`). Therefore also user level security and access is implemented in this central place. If code would be spreaded everywhere such as in a GUI this will be more difficult. The other advantage is that any code you add to the Director is automatically available to all interface programs, like the tty console and other programs.

11.1.2 GUI Interface is Difficult

Interfacing to an interactive program such as Bareos can be very difficult because the interfacing program must interpret all the prompts that may come. This can be next to impossible. There are are a number of ways that Bareos is designed to facilitate this:

- The Bareos network protocol is packet based, and thus pieces of information sent can be ASCII or binary.
- The packet interface permits knowing where the end of a list is.
- The packet interface permits special "signals" to be passed rather than data.
- The Director has a number of commands that are non-interactive. They all begin with a period, and provide things such as the list of all Jobs, list of all Clients, list of all Pools, list of all Storage, ... Thus the GUI interface can get to virtually all information that the Director has in a deterministic way. See https://github.com/bareos/bareos/blob/bareos-17.2/src/dird/ua_dotcmds.c for more details on this.
- Most console commands allow all the arguments to be specified on the command line: e.g. `run job=NightlyBackup level=Full`

One of the first things to overcome is to be able to establish a conversation with the Director. Although you can write all your own code, it is probably easier to use the Bareos subroutines. The following code is used by the Console program to begin a conversation.

```
static BSOCK *UA_sock = NULL;
static JCR *jcr;
...
  read-your-config-getting-address-and-pasword;
  UA_sock = bnet_connect(NULL, 5, 15, "Director daemon", dir->address,
                         NULL, dir->DIRport, 0);
   if (UA_sock == NULL) {
```

```
      terminate_console(0);
      return 1;
   }
   jcr.dir_bsock = UA_sock;
   if (!authenticate_director(\&jcr, dir)) {
      fprintf(stderr, "ERR=%s", UA_sock->msg);
      terminate_console(0);
      return 1;
   }
   read_and_process_input(stdin, UA_sock);
   if (UA_sock) {
      bnet_sig(UA_sock, BNET_TERMINATE); /* send EOF */
      bnet_close(UA_sock);
   }
   exit 0;
```

Then the read_and_process_input routine looks like the following:

```
get-input-to-send-to-the-Director;
bnet_fsend(UA_sock, "%s", input);
stat = bnet_recv(UA_sock);
process-output-from-the-Director;
```

For a GUI program things will be a bit more complicated. Basically in the very inner loop, you will need to check and see if any output is available on the UA_sock.

11.2 dot commands

Besides the normal commands (like list, status, run, mount, ...) the Director offers a number of so called *dot commands*. They all begin with a period, are all non-interactive, easily parseable and are indended to be used by other Bareos interface programs (GUIs).

See https://github.com/bareos/bareos/blob/master/src/dird/ua_dotcmds.c for more details.

- `.actiononpurge`
- `.api [ 0 | 1 | 2 | off | on | json ]`
  - Switch between different *api modes*
- `.clients`
  - List all client resources
- `.catalogs`
  - List all catalog resources
- `.defaults job=<job-name> | client=<client-name> | storage=<storage-name | pool=<pool-name>`
  - List default settings
- `.filesets`
  - List all filesets
- `.help [ all | item=cmd ]`
  - Print parsable information about a command

- `.jobdefs`
  - List add JobDef resources
- `.jobs`
  - List job resources
- `.levels`
  - List all backup levels
- `.locations`
- `.messages`
- `.media`
  - List all medias
- `.mediatypes`
  - List all media types
- `.msgs`
  - List all message resources
- `.pools`
  - List all pool resources
- `.profiles`
  - List all profile resources
- `.quit`
  - Close connection
- `.sql query=<sqlquery>`
  - Send an arbitary SQL command
- `.schedule`
  - List all schedule resources
- `.status`
- `.storages`
  - List all storage resources
- `.types`
  - List all job types
- `.volstatus`
  - List all volume status
- `.bvfs_lsdirs`
- `.bvfs_lsfiles`
- `.bvfs_update`
- `.bvfs_get_jobids`
- `.bvfs_versions`

- `.bvfs_restore`
- `.bvfs_cleanup`
- `.bvfs_clear_cache`

11.3 API Modes

The `.api` command can be used to switch between the different API modes. Besides the `.api` command, there is also the `gui on | off` command. However, this command can be ignored, as it set to gui on in command execution anyway.

11.3.1 API mode 0 (off)

```
.api 0
```

By default, a console connection to the Director is in interactive mode, meaning the api mode is off. This is the normal mode you get when using the bconsole. The output should be human readable.

11.3.2 API mode 1 (on)

To get better parsable output, a console connection could be switched to API mode 1 (on).

```
.api 1
```

or (form times where they have only been one API flavour)

```
.api
```

This mode is intended to create output that is earlier parsable. Internaly some commands vary there output for the API mode 1, but not all.

In API mode 1 some output is only delimted by the end of a packet, by not a new line. bconsole does not display end of packets (for good reason, as some output (e.g. `status`) is send in multiple packets). If running in a bconsole, this leads not parsable output for human.

Example:

```
*.api 0
api: 0
*.defaults job=BackupClient1
job=BackupClient1
pool=Incremental
messages=Standard
client=client1.example.com-fd
storage=File
where=
level=Incremental
type=Backup
fileset=SelfTest
enabled=1
catalog=MyCatalog
*.api 1
api: 1
```

```
*.defaults job=BackupClient1
job=BackupClient1pool=Incrementalmessages=Standardclient=client1.example.com-
↪fdstorage=Filewhere=level=Incrementaltype=Backupfileset=SelfTestenabled=1catalog=MyCatalog
```

This mode is used by BAT.

- *Signals*

11.3.3 API mode 2 (json)

The API mode 2 (or JSON mode) has been introduced in Bareos-15.2 and differs from API mode 1 in several aspects:

- JSON output
- The JSON output is in the format of JSON-RPC 2.0 responce objects (http://www.jsonrpc.org/specification#response_object). This should make it easier to implement a full JSON-RPC service later.
- No user interaction inside a command (meaning: if not all parameter are given to a `run` command, the command fails).
- Each command creates exaclty one responce object.

Currently a subset of the available commands return there result in JSON format, while others still write plain text output. When finished, it should be safe to run all commands in JSON mode.

A successful responce should return

```
"result": {
    "<type_of_the_results>": [
        {
            <result_object_1_key_1>: <result_object_1_value_1>,
            <result_object_1_key_2>: <result_object_1_value_2>,
            ...
        },
        {
            <result_object_2_key_1>: <result_object_2_value_1>,
            <result_object_2_key_2>: <result_object_2_value_2>,
            ...
        },
        ...
    ]
}
```

All keys are lower case.

Examples

- list
  - e.g.

```
*list jobs
{
"jsonrpc": "2.0",
"id": null,
"result": {
  "jobs": [
    {
```

```
      "type": "B",
      "starttime": "2015-06-25 16:51:38",
      "jobfiles": "18",
      "jobid": "1",
      "name": "BackupClient1",
      "jobstatus": "T",
      "level": "F",
      "jobbytes": "4651943"
    },
    {
      "type": "B",
      "starttime": "2015-06-25 17:25:23",
      "jobfiles": "0",
      "jobid": "2",
      "name": "BackupClient1",
      "jobstatus": "T",
      "level": "I",
      "jobbytes": "0"
    },
    ...
  ]
}
}
```


  - keys are the table names

- llist
  - e.g.


```
*llist jobs
{
"jsonrpc": "2.0",
"id": null,
"result": {
  "jobs": [
    {
      "name": "BackupClient1",
      "realendtime": "2015-06-25 16:51:40",
      "Type": "B",
      "schedtime": "2015-06-25 16:51:33",
      "poolid": "1",
      "level": "F",
      "jobfiles": "18",
      "volsessionid": "1",
      "jobid": "1",
      "job": "BackupClient1.2015-06-25_16.51.35_04",
      "priorjobid": "0",
      "endtime": "2015-06-25 16:51:40",
      "jobtdate": "1435243900",
      "jobstatus": "T",
      "jobmissingfiles": "0",
      "joberrors": "0",
      "purgedfiles": "0",
      "starttime": "2015-06-25 16:51:38",
      "clientname": "ting.dass-it-fd",
      "clientid": "1",
      "volsessiontime": "1435243839",
      "filesetid": "1",
```

```
      "poolname": "Full",
      "fileset": "SelfTest"
    },
    {
      "name": "BackupClient1",
      "realendtime": "2015-06-25 17:25:24",
      "type": "B",
      "schedtime": "2015-06-25 17:25:10",
      "poolid": "3",
      "level": "I",
      "jobfiles": "0",
      "volsessionid": "2",
      "jobid": "2",
      "job": "BackupClient1.2015-06-25_17.25.20_04",
      "priorjobid": "0",
      "endtime": "2015-06-25 17:25:24",
      "jobtdate": "1435245924",
      "jobstatus": "T",
      "jobmissingfiles": "0",
      "JobErrors": "0",
      "purgedfiles": "0",
      "starttime": "2015-06-25 17:25:23",
      "clientname": "ting.dass-it-fd",
      "clientid": "1",
      "volsessiontime": "1435243839",
      "filesetid": "1",
      "poolname": "Incremental",
      "fileset": "SelfTest"
    },
    ...
  ]
}
}
```


  - like the list `command`, but more values

- .jobs
  - e.g.


```
*.jobs
{
"jsonrpc": "2.0",
"id": null,
"result": {
  "jobs": [
    {
      "name": "BackupClient1"
    },
    {
      "name": "BackupCatalog"
    },
    {
      "name": "RestoreFiles"
    }
  ]
}
}
```

Example of a JSON-RPC Error Response

Example of a JSON-RPC Error Response (http://www.jsonrpc.org/specification#error_object):

```
*gui
{
  "jsonrpc": "2.0",
  "id": null,
  "error": {
    "data": {
      "result": {},
      "messages": {
        "error": [
          "ON or OFF keyword missing.\n"
        ]
      }
    },
    "message": "failed",
    "code": 1
  }
}
```

- an error response is emitted, if the command returns false or emitted an error message (`void UAContext::error_msg(const char *fmt,...)`). Messages and the result so far will be part of the error response object.

11.4 Bvfs API

The BVFS (Bareos Virtual File System) do provide a API for browsing the backuped files in the catalog and select files for restoring.

The Bvfs module works correctly with BaseJobs, Copy and Migration jobs.

The initial version in Bacula have be founded by Bacula Systems.

11.4.1 General notes

- All fields are separated by a tab (api mode 0 and 1). (api mode 2: JSON format).
- The output format for api mode 0 and 1 have changed for bareos >= 17.2. In earlier versions the second column of the `bvfs_lsdirs`, `bvfs_lsfiles` and `bvfs_versions` command have been the `FilenameId`. As bareos >= 17.2 internally don't use the `FilenameId` any longer, this column have been removed.
- You can specify `limit=` and `offset=` to list smoothly records in very big directories. By default, limit=2000.
- All operations (except cache creation) are designed to run instantly.
- The cache creation is dependent of the number of directories. As Bvfs shares information across jobs, the first creation can be slow.
- Due to potential encoding problem, it's advised to use `pathid` instead of `path` in queries.

11.4.2 Get dependent jobs from a given JobId

Bvfs allows you to query the catalog against any combination of jobs. You can combine all Jobs and all FileSet for a Client in a single session.

To get all JobId needed to restore a particular job, you can use the `.bvfs_get_jobids` command.

```
.bvfs_get_jobids jobid=num [all]
```

Example:

```
*.bvfs_get_jobids jobid=10
1,2,5,10
*.bvfs_get_jobids jobid=10 all
1,2,3,5,10
```

In this example, a normal restore will need to use JobIds 1,2,5,10 to compute a complete restore of the system.

With the `all` option, the Director will use all defined FileSet for this client.

11.4.3 Generating Bvfs cache

The `.bvfs_update` command computes the directory cache for jobs specified in argument, or for all jobs if unspecified.

```
.bvfs_update [jobid=numlist]
```

Example:

```
*.bvfs_update jobid=1,2,3
```

You can run the cache update process in a RunScript after the catalog backup.

11.4.4 List directories

Bvfs allows you to list directories in a specific path.

```
*.bvfs_lsdirs pathid=num path=/apath jobid=numlist limit=num offset=num
PathId  FileId  JobId  LStat  Path
PathId  FileId  JobId  LStat  Path
PathId  FileId  JobId  LStat  Path
...
```

In bareos < 17.2 the output has been:

```
PathId  FilenameId  FileId  JobId  LStat  Path
```

You need to `pathid` or `path`. Using `path=` will list "/" on Unix and all drives on Windows.

FilenameId is 0 for all directories.

```
*.bvfs_lsdirs pathid=4 jobid=1,11,12
4       0       0       A A A A A A A A A A A A A A     .
5       0       0       A A A A A A A A A A A A A A     ..
3       0       0       A A A A A A A A A A A A A A     regress/
```

In this example, to list directories present in `regress/`, you can use

```
*.bvfs_lsdirs pathid=3 jobid=1,11,12
3       0       0       A A A A A A A A A A A A A A     .
4       0       0       A A A A A A A A A A A A A A     ..
2       0       0       A A A A A A A A A A A A A A     tmp/
```

11.4.5 List files

API mode 0

Bvfs allows you to list files in a specific path.

```
.bvfs_lsfiles pathid=num path=/apath jobid=numlist limit=num offset=num
PathId  FileId  JobId  LStat  Filename
PathId  FileId  JobId  LStat  Filename
PathId  FileId  JobId  LStat  Filename
...
```

In bareos < 17.2 the output has been:

```
PathId  FilenameId  FileId  JobId  LStat  Filename
```

You need to `pathid` or `path`. Using `path=` will list "/" on Unix and all drives on Windows. If FilenameId is 0, the record listed is a directory.

```
*.bvfs_lsdir pathid=4 jobid=1,11,12
4       0       0       A A A A A A A A A A A A A A     .
5       0       0       A A A A A A A A A A A A A A     ..
1       0       0       A A A A A A A A A A A A A A     regress/
```

In this example, to list files present in `regress/`, you can use

```
*.bvfs_lsfiles pathid=1 jobid=1,11,12
1   52   12    gD HRid IGk BAA I BMqcPH BMqcPE BMqe+t A     titi
1   53   12    gD HRid IGk BAA I BMqe/K BMqcPE BMqe+t B     toto
1   54   12    gD HRie IGk BAA I BMqcPH BMqcPE BMqe+3 A     tutu
1   55   12    gD HRid IGk BAA I BMqe/K BMqcPE BMqe+t B     ficheriro1.txt
1   56   12    gD HRie IGk BAA I BMqe/K BMqcPE BMqe+3 D     ficheriro2.txt
```

API mode 1

```
*.api 1
*.bvfs_lsfiles jobid=1 pathid=1
1   7   1   gD OEE4 IHo B GHH GHH A G9S BAA 4 BVjBQG BVjBQG BVjBQG A A C    bpluginfo
1   4   1   gD OEE3 KH/ B GHH GHH A W BAA A BVjBQ7 BVjBQG BVjBQG A A C  bregex
...
```

API mode 2

```
*.api 2
*.bvfs_lsfiles jobid=1 pathid=1
{
  "jsonrpc": "2.0",
  "id": null,
  "result": {
    "files": [
      {
```

```
        "jobid": 1,
        "type": "F",
        "fileid": 7,
        "lstat": "gD OEE4 IHo B GHH GHH A G9S BAA 4 BVjBQG BVjBQG BVjBQG A A C",
        "pathid": 1,
        "stat": {
          "atime": 1435243526,
          "ino": 3686712,
          "dev": 2051,
          "mode": 33256,
          "gid": 25031,
          "nlink": 1,
          "uid": 25031,
          "ctime": 1435243526,
          "rdev": 0,
          "size": 28498,
          "mtime": 1435243526
        },
        "name": "bpluginfo",
        "linkfileindex": 0
      },
      {
        "jobid": 1,
        "type": "F",
        "fileid": 4,
        "lstat": "gD OEE3 KH/ B GHH GHH A W BAA A BVjBQ7 BVjBQG BVjBQG A A C",
        "pathid": 1,
        "stat": {
          "atime": 1435243579,
          "ino": 3686711,
          "dev": 2051,
          "mode": 41471,
          "gid": 25031,
          "nlink": 1,
          "uid": 25031,
          "ctime": 1435243526,
          "rdev": 0,
          "size": 22,
          "mtime": 1435243526
        },
        "name": "bregex",
        "linkfileindex": 0
      },
      ...
    ]
  }
}
```


API mode JSON contains all information also available in the other API modes, but displays them more verbose.

11.4.6 Get all versions of a specific file

Bvfs allows you to find all versions of a specific file for a given Client with the `.bvfs_version` command. To avoid problems with encoding, this function uses only PathId and FilenameId.

The jobid argument is mandatory but unused.

```
*.bvfs_versions jobid=0 client=filedaemon pathid=num fname=filename [copies]␣
↪[versions]
PathId FileId JobId LStat Md5 VolName InChanger
PathId FileId JobId LStat Md5 VolName InChanger
...
```

Example:

```
*.bvfs_versions jobid=0 client=localhost-fd pathid=1 fnane=toto
1  49  12  gD HRid IGk D Po Po A P BAA I A   /uPgWaxMgKZlnMti7LChyA  Vol1  1
```

11.4.7 Restore set of files

Bvfs allows you to create a SQL table that contains files that you want to restore. This table can be provided to a restore command with the file option.

```
*.bvfs_restore fileid=numlist dirid=numlist hardlink=numlist path=b2num
OK
*restore file=?b2num ...
```

To include a directory (with `dirid`), Bvfs needs to run a query to select all files. This query could be time consuming.

`hardlink` list is always composed of a serie of two numbers (jobid, fileindex). This information can be found in the LinkFileIndex (LinkFI) field of the LStat packet.

The `path` argument represents the name of the table that Bvfs will store results. The format of this table is `b2[0-9]+`. (Should start by b2 and followed by digits).

Example:

```
*.bvfs_restore fileid=1,2,3,4 hardlink=10,15,10,20 jobid=10 path=b20001
OK
```

11.4.8 Cleanup after Restore

To drop the table used by the restore command, you can use the `.bvfs_cleanup` command.

```
*.bvfs_cleanup path=b20001
```

11.4.9 Clearing the BVFS Cache

To clear the BVFS cache, you can use the `.bvfs_clear_cache` command.

```
*.bvfs_clear_cache yes
OK
```

11.4.10 Example for directory browsing using bvfs

```
# update the bvfs cache for all jobs
*.bvfs_update
Automatically selected Catalog: MyCatalog
```

```
Using Catalog "MyCatalog

# Get jobids required to reconstruct a current full backup.
# This is optional. Only required if you care about a full backup.
# If you are only interessed in a single (differential or incremental) backup job,
# just use the single jobid.
*.bvfs_get_jobids jobid=123
117,118,123

# get root directory of the combined jobs 117,118,123
*.bvfs_lsdir jobid=117,118,123 path=
134 0   0   A A A A A A A A A A A A A .
133 0   0   A A A A A A A A A A A A A /

# path=/ (pathid=133) is the root directory.
# Check the root directory for subdirectories.
.bvfs_lsdir jobid=117,118,123 pathid=133
133 0   0   A A A A A A A A A A A A A .
130 0   0   A A A A A A A A A A A A A ..
1   23  123 z GiuU EH9 C GHH GHH A BAA BAA I BWA5Px BaIDUN BaIDUN A A C sbin/

# the first really backuped path is /sbin/ (pathid=1)
# as it has values other than 0 for FileId, JobId and LStat.
# Now we check, if it has futher subdirectories.
*.bvfs_lsdir jobid=1 pathid=1
1   23  123 z GiuU EH9 C GHH GHH A BAA BAA I BWA5Px BaIDUN BaIDUN A A C .
129 0   0   A A A A A A A A A A A A A ..

# pathid=1 has no further subdirectories.
# Now we list the files in pathid=1 (/sbin/)
.bvfs_lsfiles jobid=117,118,123 pathid=1
1   18  123 z Gli+ IHo B GHH GHH A NVkY BAA BrA BaIDUJ BaIDUJ BaIDUJ A A C  bareos-dir
1   21  123 z GkuS IHo B GHH GHH A C1bw BAA XA BaIDUG BaIDUG BaIDUG A A C   bareos-fd
1   19  123 z Glju IHo B GHH GHH A CeNg BAA UI BaIDUJ BaIDUJ BaIDUJ A A C   bareos-sd
...

# there are a number of files in /sbin/.
# We check, if there are different versions of the file bareos-dir.
*.bvfs_versions jobid=0 client=bareos-fd pathid=1 fname=bareos-dir
1   18  123 z Gli+ IHo B GHH GHH A NVkY BAA BrA BaIDUJ BaIDUJ BaIDUJ A A C ␣
↪928EB+EJGFtWD7wQ8bVjew  Full-0001   0
1   1067    127 z Glnc IHo B GHH GHH A NVkY BAA BrA BaKDT2 BaKDT2 BaKDT2 A A C ␣
↪928EB+EJGFtWD7wQ8bVjew  Incremental-0007    0

# multiple versions of the file bareos-dir have been backuped.
```

CHAPTER

TWELVE

TLS

Written by Landon Fuller

12.1 Introduction to TLS

This patch includes all the back-end code necessary to add complete TLS data encryption support to Bareos. In addition, support for TLS in Console/Director communications has been added as a proof of concept. Adding support for the remaining daemons will be straight-forward. Supported features of this patchset include:

- Client/Server TLS Requirement Negotiation
- TLSv1 Connections with Server and Client Certificate Validation
- Forward Secrecy Support via Diffie-Hellman Ephemeral Keying

This document will refer to both "server" and "client" contexts. These terms refer to the accepting and initiating peer, respectively.

Diffie-Hellman anonymous ciphers are not supported by this patchset. The use of DH anonymous ciphers increases the code complexity and places explicit trust upon the two-way Cram-MD5 implementation. Cram-MD5 is subject to known plaintext attacks, and is should be considered considerably less secure than PKI certificate-based authentication.

12.2 TLS API Implementation

Appropriate autoconf macros have been added to detect and use OpenSSL. Two additional preprocessor defines have been added: `HAVE_TLS` and `HAVE_OPENSSL`. All changes not specific to OpenSSL rely on `HAVE_TLS`. In turn, a generic TLS API is exported.

12.2.1 Library Initialization and Cleanup

```
int init_tls(void);
```

Performs TLS library initialization, including seeding of the PRNG. PRNG seeding has not yet been implemented for win32.

```
int cleanup_tls(void);
```

Performs TLS library cleanup.

12.2.2 Manipulating TLS Contexts

```
TLS_CONTEXT  *new_tls_context(const char *ca_certfile,
        const char *ca_certdir, const char *certfile,
        const char *keyfile, const char *dhfile, bool verify_peer);
```

Allocates and initalizes a new opaque *TLS_CONTEXT* structure. The *TLS_CONTEXT* structure maintains default TLS settings from which *TLS_CONNECTION* structures are instantiated. In the future the *TLS_CONTEXT* structure may be used to maintain the TLS session cache. *ca_certfile* and *ca_certdir* arguments are used to initialize the CA verification stores. The *certfile* and *keyfile* arguments are used to initialize the local certificate and private key. If *dhfile* is non-NULL, it is used to initialize Diffie-Hellman ephemeral keying. If *verify_peer* is *true* , client certificate validation is enabled.

```
void free_tls_context(TLS_CONTEXT *ctx);
```

Deallocated a previously allocated *TLS_CONTEXT* structure.

12.2.3 Performing Post-Connection Verification

```
bool tls_postconnect_verify_host(TLS_CONNECTION *tls, const char *host);
```

Performs post-connection verification of the peer-supplied x509 certificate. Checks whether the *subjectAltName* and *commonName* attributes match the supplied *host* string. Returns *true* if there is a match, *false* otherwise.

```
bool tls_postconnect_verify_cn(TLS_CONNECTION *tls, alist *verify_list);
```

Performs post-connection verification of the peer-supplied x509 certificate. Checks whether the *commonName* attribute matches any strings supplied via the *verify_list* parameter. Returns *true* if there is a match, *false* otherwise.

12.2.4 Manipulating TLS Connections

```
TLS_CONNECTION *new_tls_connection(TLS_CONTEXT *ctx, int fd);
```

Allocates and initializes a new *TLS_CONNECTION* structure with context *ctx* and file descriptor *fd*.

```
void free_tls_connection(TLS_CONNECTION *tls);
```

Deallocates memory associated with the *tls* structure.

```
bool tls_bsock_connect(BSOCK *bsock);
```

Negotiates a a TLS client connection via *bsock*. Returns *true* if successful, *false* otherwise. Will fail if there is a TLS protocol error or an invalid certificate is presented

```
bool tls_bsock_accept(BSOCK *bsock);
```

Accepts a TLS client connection via *bsock*. Returns *true* if successful, *false* otherwise. Will fail if there is a TLS protocol error or an invalid certificate is presented.

```
bool tls_bsock_shutdown(BSOCK *bsock);
```

Issues a blocking TLS shutdown request to the peer via *bsock*. This function may not wait for the peer's reply.

```
int tls_bsock_writen(BSOCK *bsock, char *ptr, int32_t nbytes);
```

Writes *nbytes* from *ptr* via the *TLS_CONNECTION* associated with *bsock*. Due to OpenSSL's handling of *EINTR*, *bsock* is set non-blocking at the start of the function, and restored to its original blocking state before the function returns. Less than *nbytes* may be written if an error occurs. The actual number of bytes written will be returned.

```
int tls_bsock_readn(BSOCK *bsock, char *ptr, int32_t nbytes);
```

Reads *nbytes* from the *TLS_CONNECTION* associated with *bsock* and stores the result in *ptr*. Due to OpenSSL's handling of *EINTR*, *bsock* is set non-blocking at the start of the function, and restored to its original blocking state before the function returns. Less than *nbytes* may be read if an error occurs. The actual number of bytes read will be returned.

12.3 Bnet API Changes

A minimal number of changes were required in the Bnet socket API. The BSOCK structure was expanded to include an associated TLS_CONNECTION structure, as well as a flag to designate the current blocking state of the socket. The blocking state flag is required for win32, where it does not appear possible to discern the current blocking state of a socket.

12.3.1 Negotiating a TLS Connection

bnet_tls_server() and *bnet_tls_client()* were both implemented using the new TLS API as follows:

```
int bnet_tls_client(TLS_CONTEXT *ctx, BSOCK * bsock);
```

Negotiates a TLS session via *bsock* using the settings from *ctx*. Returns 1 if successful, 0 otherwise.

```
int bnet_tls_server(TLS_CONTEXT *ctx, BSOCK * bsock, alist *verify_list);
```

Accepts a TLS client session via *bsock* using the settings from *ctx*. If *verify_list* is non-NULL, it is passed to *tls_postconnect_verify_cn()* for client certificate verification.

12.3.2 Manipulating Socket Blocking State

Three functions were added for manipulating the blocking state of a socket on both Win32 and Unix-like systems. The Win32 code was written according to the MSDN documentation, but has not been tested.

These functions are prototyped as follows:

```
int bnet_set_nonblocking(BSOCK *bsock);
```

Enables non-blocking I/O on the socket associated with *bsock*. Returns a copy of the socket flags prior to modification.

```
int bnet_set_blocking(BSOCK *bsock);
```

Enables blocking I/O on the socket associated with *bsock*. Returns a copy of the socket flags prior to modification.

```
void bnet_restore_blocking(BSOCK *bsock, int flags);
```

Restores blocking or non-blocking IO setting on the socket associated with *bsock*. The *flags* argument must be the return value of either *bnet_set_blocking()* or *bnet_restore_blocking()*.

12.4 Authentication Negotiation

Backwards compatibility with the existing SSL negotiation hooks implemented in src/lib/cram-md5.c have been maintained. The *cram_md5_get_auth()* function has been modified to accept an integer pointer argument, tls_remote_need. The TLS requirement advertised by the remote host is returned via this pointer.

After exchanging cram-md5 authentication and TLS requirements, both the client and server independently decide whether to continue:

```
if (!cram_md5_get_auth(dir, password, &tls_remote_need) ||
        !cram_md5_auth(dir, password, tls_local_need)) {
[snip]
/* Verify that the remote host is willing to meet our TLS requirements */
if (tls_remote_need < tls_local_need && tls_local_need != BNET_TLS_OK &&
        tls_remote_need != BNET_TLS_OK) {
   sendit(_("Authorization problem:"
            " Remote server did not advertise required TLS support.\n"));
   auth_success = false;
   goto auth_done;
}

/* Verify that we are willing to meet the remote host's requirements */
if (tls_remote_need > tls_local_need && tls_local_need != BNET_TLS_OK &&
        tls_remote_need != BNET_TLS_OK) {
   sendit(_("Authorization problem:"
            " Remote server requires TLS.\n"));
   auth_success = false;
   goto auth_done;
}
```

CHAPTER

THIRTEEN

BAREOS REGRESSION TESTING

13.1 Setting up Regession Testing

This document is intended mostly for developers who wish to ensure that their changes to Bareos don't introduce bugs in the base code. However, you don't need to be a developer to run the regression scripts, and we recommend them before putting your system into production, and before each upgrade, especially if you build from source code. They are simply shell scripts that drive Bareos through bconsole and then typically compare the input and output with **diff**.

You can find the existing regression scripts in the Bareos developer's **git** repository on SourceForge. We strongly recommend that you **clone** the repository because afterwards, you can easily get pull the updates that have been made.

To get started, we recommend that you create a directory named **bareos**, under which you will put the current source code and the current set of regression scripts. Below, we will describe how to set this up.

The top level directory that we call **bareos** can be named anything you want. Note, all the standard regression scripts run as non-root and can be run on the same machine as a production Bareos system (the developers run it this way).

To create the directory structure for the current trunk and to clone the repository, do the following (note, we assume you are working in your home directory in a non-root account):

```
git clone https://github.com/bareos/bareos-regress.git
```

This will create the directory **bareos-regress**. The above should be needed only once. Thereafter to update to the latest code, you do:

```
cd bareos-regress
git pull
```

There are two different aspects of regression testing that this document will discuss: 1. Running the Regression Script, 2. Writing a Regression test.

13.2 Running the Regression Script

There are a number of different tests that may be run, such as: the standard set that uses disk Volumes and runs under any userid; a small set of tests that write to tape; another set of tests where you must be root to run them. Normally, I run all my tests as non-root and very rarely run the root tests. The tests vary in length, and running the full tests including disk based testing, tape based testing, autochanger based testing, and multiple drive autochanger based testing can take 3 or 4 hours.

13.2.1 Setting the Configuration Parameters

There is nothing you need to change in the source directory.

To begin:

```
cd bareos-regress
```

The very first time you are going to run the regression scripts, you will need to create a custom config file for your system. We suggest that you start by:

```
cp prototype.conf config
```

Then you can edit the **config** file directly.

```
# Where to get the source to be tested
BAREOS_SOURCE="${HOME}/bareos/bareos"

# Where to send email   !!!!! Change me !!!!!!!
EMAIL=your-name@your-domain.com
SMTP_HOST="localhost"

TAPE_DRIVE="/dev/nst0"
# if you don't have an autochanger set AUTOCHANGER to /dev/null
AUTOCHANGER="/dev/sg0"
# For two drive tests -- set to /dev/null if you do not have it
TAPE_DRIVE1="/dev/null"

# This must be the path to the autochanger including its name
AUTOCHANGER_PATH="/usr/sbin/mtx"

# Set what backend to use "postresql" "mysql" or "sqlite3"
DBTYPE="postgresql"

# Set your database here
#WHICHDB="--with-${DBTYPE}=${SQLITE3_DIR}"
WHICHDB="--with-${DBTYPE}"

# Set this to "--with-tcp-wrappers" or "--without-tcp-wrappers"
TCPWRAPPERS="--with-tcp-wrappers"

# Set this to "" to disable OpenSSL support, "--with-openssl=yes"
# to enable it, or provide the path to the OpenSSL installation,
# eg "--with-openssl=/usr/local"
OPENSSL="--with-openssl"

# You may put your real host name here, but localhost or 127.0.0.1
# is valid also and it has the advantage that it works on a
# non-networked machine
HOST="localhost"
```

- **BAREOS_SOURCE** should be the full path to the Bareos source code that you wish to test. It will be loaded configured, compiled, and installed with the "make setup" command, which needs to be done only once each time you change the source code.
- **EMAIL** should be your email addres. Please remember to change this or I will get a flood of unwanted messages. You may or may not want to see these emails. In my case, I don't need them so I direct it to the bit bucket.
- **SMTP_HOST** defines where your SMTP server is.

- **SQLITE_DIR** should be the full path to the sqlite package, must be build before running a Bareos regression, if you are using SQLite. This variable is ignored if you are using MySQL or PostgreSQL. To use PostgreSQL, edit the Makefile and change (or add) WHICHDB?="−−with-postgresql". For MySQL use "WHICHDB="−−with-mysql''.

  The advantage of using SQLite is that it is totally independent of any installation you may have running on your system, and there is no special configuration or authorization that must be done to run it. With both MySQL and PostgreSQL, you must pre-install the packages, initialize them and ensure that you have authorization to access the database and create and delete tables.

- **TAPE_DRIVE** is the full path to your tape drive. The base set of regression tests do not use a tape, so this is only important if you want to run the full tests. Set this to /dev/null if you do not have a tape drive.
- **TAPE_DRIVE1** is the full path to your second tape drive, if have one. The base set of regression tests do not use a tape, so this is only important if you want to run the full two drive tests. Set this to /dev/null if you do not have a second tape drive.
- **AUTOCHANGER** is the name of your autochanger control device. Set this to /dev/null if you do not have an autochanger.
- **AUTOCHANGER_PATH** is the full path including the program name for your autochanger program (normally **mtx**. Leave the default value if you do not have one.
- **TCPWRAPPERS** defines whether or not you want the ./configure to be performed with tcpwrappers enabled.
- **OPENSSL** used to enable/disable SSL support for Bareos communications and data encryption.
- **HOST** is the hostname that it will use when building the scripts. The Bareos daemons will be named <HOST>-dir, <HOST>-fd, ... It is also the name of the HOST machine that to connect to the daemons by the network. Hence the name should either be your real hostname (with an appropriate DNS or /etc/hosts entry) or **localhost** as it is in the default file.
- **bin** is the binary location.
- **scripts** is the bareos scripts location (where we could find database creation script, autochanger handler, etc.)

13.2.2 Building the Test Bareos

Once the above variables are set, you can build the setup by entering:

```
make setup
```

This will setup the regression testing and you should not need to do this again unless you want to change the database or other regression configuration parameters.

13.2.3 Setting up your SQL engine

If you are using SQLite or SQLite3, there is nothing more to do; you can simply run the tests as described in the next section.

If you are using MySQL or PostgreSQL, you will need to establish an account with your database engine for the user name **regress** and you will need to manually create a database named **regress** that can be used by user name regress, which means you will have to give the user regress sufficient permissions to use the database named regress. There is no password on the regress account.

You have probably already done this procedure for the user name and database named bareos. If not, the manual describes roughly how to do it, and the scripts in bareos/regress/build/src/cats named create_mysql_database, create_postgresql_database, grant_mysql_privileges, and grant_postgresql_privileges may be of a help to you.

Generally, to do the above, you will need to run under root to be able to create databases and modify permissions within MySQL and PostgreSQL.

It is possible to configure MySQL access for database accounts that require a password to be supplied. This can be done by creating a /.my.cnf file which supplies the credentials by default to the MySQL commandline utilities.

```
[client]
host     = localhost
user     = regress
password = asecret
```

A similar technique can be used PostgreSQL regression testing where the database is configured to require a password. The /.pgpass file should contain a line with the database connection properties.

```
hostname:port:database:username:password
```

13.2.4 Running the Disk Only Regression

The simplest way to copy the source code, configure it, compile it, link it, and run the tests is to use a helper script:

```
./do_disk
```

This will run the base set of tests using disk Volumes. If you are testing on a non-Linux machine several of the of the tests may not be run. In any case, as we add new tests, the number will vary. It will take about 1 hour and you don't need to be root to run these tests (I run under my regular userid). The result should be something similar to:

```
Test results
  ===== auto-label-test OK 12:31:33 =====
  ===== backup-bareos-test OK 12:32:32 =====
  ===== bextract-test OK 12:33:27 =====
  ===== bscan-test OK 12:34:47 =====
  ===== bsr-opt-test OK 12:35:46 =====
  ===== compressed-test OK 12:36:52 =====
  ===== compressed-encrypt-test OK 12:38:18 =====
  ===== concurrent-jobs-test OK 12:39:49 =====
  ===== data-encrypt-test OK 12:41:11 =====
  ===== encrypt-bug-test OK 12:42:00 =====
  ===== fifo-test OK 12:43:46 =====
  ===== backup-bareos-fifo OK 12:44:54 =====
  ===== differential-test OK 12:45:36 =====
  ===== four-concurrent-jobs-test OK 12:47:39 =====
  ===== four-jobs-test OK 12:49:22 =====
  ===== incremental-test OK 12:50:38 =====
  ===== query-test OK 12:51:37 =====
  ===== recycle-test OK 12:53:52 =====
  ===== restore2-by-file-test OK 12:54:53 =====
  ===== restore-by-file-test OK 12:55:40 =====
  ===== restore-disk-seek-test OK 12:56:29 =====
  ===== six-vol-test OK 12:57:44 =====
  ===== span-vol-test OK 12:58:52 =====
  ===== sparse-compressed-test OK 13:00:00 =====
  ===== sparse-test OK 13:01:04 =====
  ===== two-jobs-test OK 13:02:39 =====
  ===== two-vol-test OK 13:03:49 =====
  ===== verify-vol-test OK 13:04:56 =====
  ===== weird-files2-test OK 13:05:47 =====
  ===== weird-files-test OK 13:06:33 =====
```

```
  ===== migration-job-test OK 13:08:15 =====
  ===== migration-jobspan-test OK 13:09:33 =====
  ===== migration-volume-test OK 13:10:48 =====
  ===== migration-time-test OK 13:12:59 =====
  ===== hardlink-test OK 13:13:50 =====
  ===== two-pool-test OK 13:18:17 =====
  ===== fast-two-pool-test OK 13:24:02 =====
  ===== two-volume-test OK 13:25:06 =====
  ===== incremental-2disk OK 13:25:57 =====
  ===== 2drive-incremental-2disk OK 13:26:53 =====
  ===== scratch-pool-test OK 13:28:01 =====
Total time = 0:57:55 or 3475 secs
```

and the working tape tests are run with

```
make full_test

Test results

  ===== Bareos tape test OK =====
  ===== Small File Size test OK =====
  ===== restore-by-file-tape test OK =====
  ===== incremental-tape test OK =====
  ===== four-concurrent-jobs-tape OK =====
  ===== four-jobs-tape OK =====
```

Each separate test is self contained in that it initializes to run Bareos from scratch (i.e. newly created database). It will also kill any Bareos session that is currently running. In addition, it uses ports 8101, 8102, and 8103 so that it does not intefere with a production system.

Alternatively, you can do the ./do_disk work by hand with:

```
make setup
```

The above will then copy the source code within the regression tree (in directory regress/build), configure it, and build it. There should be no errors. If there are, please correct them before continuing. From this point on, as long as you don't change the Bareos source code, you should not need to repeat any of the above steps. If you pull down a new version of the source code, simply run **make setup** again.

Once Bareos is built, you can run the basic disk only non-root regression test by entering:

```
make test
```

13.2.5 Other Tests

There are a number of other tests that can be run as well. All the tests are a simply shell script keep in the regress directory. For example the "make test'' simply executes **./all-non-root-tests**. The other tests, which are invoked by directly running the script are:

all_non-root-tests All non-tape tests not requiring root. This is the standard set of tests, that in general, backup some data, then restore it, and finally compares the restored data with the original data.

all-root-tests All non-tape tests requiring root permission. These are a relatively small number of tests that require running as root. The amount of data backed up can be quite large. For example, one test backs up /usr, another backs up /etc. One or more of these tests reports an error – I'll fix it one day.

all-non-root-tape-tests All tape test not requiring root. There are currently three tests, all run without being root, and backup to a tape. The first two tests use one volume, and the third test requires an autochanger, and uses two volumes. If you don't have an autochanger, then this script will probably produce an error.

all-tape-and-file-tests All tape and file tests not requiring root. This includes just about everything, and I don't run it very often.

13.2.6 If a Test Fails

If you one or more tests fail, the line output will be similar to:

```
!!!!! concurrent-jobs-test failed!!! !!!!!
```

If you want to determine why the test failed, you will need to rerun the script with the debug output turned on. You do so by defining the environment variable **REGRESS_DEBUG** with commands such as:

```
REGRESS_DEBUG=1
export REGRESS_DEBUG
```

Then from the "regress" directory (all regression scripts assume that you have "regress" as the current directory), enter:

```
tests/test-name
```

where test-name should be the name of a test script – for example: **tests/backup-bareos-test**.

13.3 Testing a Binary Installation

If you have installed your Bareos from a binary release such as (rpms or debs), you can still run regression tests on it. First, make sure that your regression **config** file uses the same catalog backend as your installed binaries. Then define the variables `bin` and `scripts` variables in your config file.

Example:

```
bin=/usr/sbin/
scripts=/usr/lib/bareos/scripts/
```

The `./scripts/prepare-other-loc` will tweak the regress scripts to use your binary location. You will need to run it manually once before you run any regression tests.

```
$ ./scripts/prepare-other-loc
$ ./tests/backup-bareos-test
...
```

All regression scripts must be run by hand or by calling the test scripts. These are principally scripts that begin with **all_...** such as **all_disk_tests**, **./all_test** ...

None of the **./do_disk**, **./do_all**, **./nightly...** scripts will work.

If you want to switch back to running the regression scripts from source, first remove the **bin** and **scripts** variables from your **config** file and rerun the `make setup` step.

13.4 Running a Single Test

If you wish to run a single test, you can simply:

```
cd regress
tests/<name-of-test>
```

or, if the source code has been updated, you would do:

```
cd bareos
git pull
cd regress
make setup
tests/backup-to-null
```

13.5 Writing a Regression Test

Any developer, who implements a major new feature, should write a regression test that exercises and validates the new feature. Each regression test is a complete test by itself. It terminates any running Bareos, initializes the database, starts Bareos, then runs the test by using the console program.

13.5.1 Running the Tests by Hand

You can run any individual test by hand by cd'ing to the **regress** directory and entering:

```
tests/<test-name>
```

13.5.2 Directory Structure

The directory structure of the regression tests is:

```
regress                - Makefile, scripts to start tests
  |------ scripts      - Scripts (and old configuration files)
  |------ tests        - All test scripts are here
  |------ configs      - configuration files (for newer tests)
  |
  |------------------ -- All directories below this point are used
  |                      for testing, but are created from the
  |                      above directories and are removed with
  |                      "make distclean"
  |
  |------ bin          - This is the install directory for
  |                       Bareos to be used testing
  |------ build        - Where the Bareos source build tree is
  |------ tmp          - Most temp files go here
  |------ working      - Bareos working directory
  |------ weird-files  - Weird files used in two of the tests.
```

13.5.3 Adding a New Test

If you want to write a new regression test, it is best to start with one of the existing test scripts, and modify it to do the new test.

When adding a new test, be extremely careful about adding anything to any of the daemons' configuration files. The reason is that it may change the prompts that are sent to the console. For example, adding a Pool means that the current

scripts, which assume that Bareos automatically selects a Pool, will now be presented with a new prompt, so the test will fail. If you need to enhance the configuration files, consider making your own versions.

13.5.4 Running a Test Under The Debugger

You can run a test under the debugger (actually run a Bareos daemon under the debugger) by first setting the environment variable **REGRESS_WAIT** with commands such as:

```
REGRESS_WAIT=1
export REGRESS_WAIT
```

Then executing the script. When the script prints the following line:

```
Start Bareos under debugger and enter anything when ready ...
```

You start the Bareos component you want to run under the debugger in a different shell window. For example:

```
cd .../regress/bin
gdb bareos-sd
(possibly set breakpoints, ...)
run -s -f
```

Then enter any character in the window with the above message. An error message will appear saying that the daemon you are debugging is already running, which is the case. You can simply ignore the error message.

CHAPTER

FOURTEEN

BAREOS MEMORY MANAGEMENT

14.1 General

This document describes the memory management routines that are used in Bareos and is meant to be a technical discussion for developers rather than part of the user manual.

Since Bareos may be called upon to handle filenames of varying and more or less arbitrary length, special attention needs to be used in the code to ensure that memory buffers are sufficiently large. There are four possibilities for memory usage within **Bareos**. Each will be described in turn. They are:

- Statically allocated memory.
- Dynamically allocated memory using malloc() and free().
- Non-pooled memory.
- Pooled memory.

14.1.1 Statically Allocated Memory

Statically allocated memory is of the form:

```
char buffer[MAXSTRING];
```

The use of this kind of memory is discouraged except when you are 100% sure that the strings to be used will be of a fixed length. One example of where this is appropriate is for **Bareos** resource names, which are currently limited to 127 characters (MAX_NAME_LENGTH). Although this maximum size may change, particularly to accommodate Unicode, it will remain a relatively small value.

14.1.2 Dynamically Allocated Memory

Dynamically allocated memory is obtained using the standard malloc() routines. As in:

```
char *buf;
buf = malloc(256);
```

This kind of memory can be released with:

```
free(buf);
```

It is recommended to use this kind of memory only when you are sure that you know the memory size needed and the memory will be used for short periods of time – that is it would not be appropriate to use statically allocated memory. An example might be to obtain a large memory buffer for reading and writing files. When **SmartAlloc** is enabled, the

memory obtained by malloc() will automatically be checked for buffer overwrite (overflow) during the free() call, and all malloc'ed memory that is not released prior to termination of the program will be reported as Orphaned memory.

14.1.3 Pooled and Non-pooled Memory

In order to facility the handling of arbitrary length filenames and to efficiently handle a high volume of dynamic memory usage, we have implemented routines between the C code and the malloc routines. The first is called "Pooled" memory, and is memory, which once allocated and then released, is not returned to the system memory pool, but rather retained in a Bareos memory pool. The next request to acquire pooled memory will return any free memory block. In addition, each memory block has its current size associated with the block allowing for easy checking if the buffer is of sufficient size. This kind of memory would normally be used in high volume situations (lots of malloc()s and free()s) where the buffer length may have to frequently change to adapt to varying filename lengths.

The non-pooled memory is handled by routines similar to those used for pooled memory, allowing for easy size checking. However, non-pooled memory is returned to the system rather than being saved in the Bareos pool. This kind of memory would normally be used in low volume situations (few malloc()s and free()s), but where the size of the buffer might have to be adjusted frequently.

Types of Memory Pool:

Currently there are three memory pool types:

- PM_NOPOOL – non-pooled memory.
- PM_FNAME – a filename pool.
- PM_MESSAGE – a message buffer pool.
- PM_EMSG – error message buffer pool.

Getting Memory:

To get memory, one uses:

```
void *get_pool_memory(pool);
```

where **pool** is one of the above mentioned pool names. The size of the memory returned will be determined by the system to be most appropriate for the application.

If you wish non-pooled memory, you may alternatively call:

```
void *get_memory(size_t size);
```

The buffer length will be set to the size specified, and it will be assigned to the PM_NOPOOL pool (no pooling).

Releasing Memory:

To free memory acquired by either of the above two calls, use:

```
void free_pool_memory(void *buffer);
```

where buffer is the memory buffer returned when the memory was acquired. If the memory was originally allocated as type PM_NOPOOL, it will be released to the system, otherwise, it will be placed on the appropriate Bareos memory pool free chain to be used in a subsequent call for memory from that pool.

Determining the Memory Size:

To determine the memory buffer size, use:

```
size_t sizeof_pool_memory(void *buffer);
```

Resizing Pool Memory:

To resize pool memory, use:

```
void *realloc_pool_memory(void *buffer);
```

The buffer will be reallocated, and the contents of the original buffer will be preserved, but the address of the buffer may change.

Automatic Size Adjustment:

To have the system check and if necessary adjust the size of your pooled memory buffer, use:

```
void *check_pool_memory_size(void *buffer, size_t new-size);
```

where **new-size** is the buffer length needed. Note, if the buffer is already equal to or larger than **new-size** no buffer size change will occur. However, if a buffer size change is needed, the original contents of the buffer will be preserved, but the buffer address may change. Many of the low level Bareos subroutines expect to be passed a pool memory buffer and use this call to ensure the buffer they use is sufficiently large.

Releasing All Pooled Memory:

In order to avoid orphaned buffer error messages when terminating the program, use:

```
void close_memory_pool();
```

to free all unused memory retained in the Bareos memory pool. Note, any memory not returned to the pool via free_pool_memory() will not be released by this call.

Pooled Memory Statistics:

For debugging purposes and performance tuning, the following call will print the current memory pool statistics:

```
void print_memory_pool_stats();
```

an example output is:

```
Pool  Maxsize  Maxused  Inuse
   0      256        0      0
   1      256        1      0
   2      256        1      0
```

CHAPTER

FIFTEEN

TCP/IP NETWORK PROTOCOL

15.1 General

This document describes the TCP/IP protocol used by Bareos to communicate between the various daemons and services. The definitive definition of the protocol can be found in src/lib/bsock.h, src/lib/bnet.c and src/lib/bnet_server.c.

Bareos's network protocol is basically a "packet oriented" protocol built on a standard TCP/IP streams. At the lowest level all packet transfers are done with read() and write() requests on system sockets. Pipes are not used as they are considered unreliable for large serial data transfers between various hosts.

Using the routines described below (bnet_open, bnet_write, bnet_recv, and bnet_close) guarantees that the number of bytes you write into the socket will be received as a single record on the other end regardless of how many low level write() and read() calls are needed. All data transferred are considered to be binary data.

15.2 bnet and Threads

These bnet routines work fine in a threaded environment. However, they assume that there is only one reader or writer on the socket at any time. It is highly recommended that only a single thread access any BSOCK packet. The exception to this rule is when the socket is first opened and it is waiting for a job to start. The wait in the Storage daemon is done in one thread and then passed to another thread for subsequent handling.

If you envision having two threads using the same BSOCK, think twice, then you must implement some locking mechanism. However, it probably would not be appropriate to put locks inside the bnet subroutines for efficiency reasons.

15.3 bnet_open

To establish a connection to a server, use the subroutine:

BSOCK *bnet_open(void *jcr, char *host, char *service, int port, int *fatal) bnet_open(), if successful, returns the Bareos sock descriptor pointer to be used in subsequent bnet_send() and bnet_read() requests. If not successful, bnet_open() returns a NULL. If fatal is set on return, it means that a fatal error occurred and that you should not repeatedly call bnet_open(). Any error message will generally be sent to the JCR.

15.4 bnet_send

To send a packet, one uses the subroutine:

int bnet_send(BSOCK *sock) This routine is equivalent to a write() except that it handles the low level details. The data to be sent is expected to be in sock->msg and be sock->msglen bytes. To send a packet, bnet_send() first writes four bytes in network byte order than indicate the size of the following data packet. It returns:

```
Returns 0 on failure
Returns 1 on success
```

In the case of a failure, an error message will be sent to the JCR contained within the bsock packet.

15.5 bnet_fsend

This form uses:

int bnet_fsend(BSOCK *sock, char *format, ...) and it allows you to send a formatted messages somewhat like fprintf(). The return status is the same as bnet_send.

15.6 is_bnet_error

Fro additional error information, you can call **is_bnet_error(BSOCK *bsock)** which will return 0 if there is no error or non-zero if there is an error on the last transmission.

15.7 is_bnet_stop

The **is_bnet_stop(BSOCK *bsock)** function will return 0 if there no errors and you can continue sending. It will return non-zero if there are errors or the line is closed (no more transmissions should be sent).

15.8 bnet_recv

To read a packet, one uses the subroutine:

int bnet_recv(BSOCK *sock) This routine is similar to a read() except that it handles the low level details. bnet_read() first reads packet length that follows as four bytes in network byte order. The data is read into sock->msg and is sock->msglen bytes. If the sock->msg is not large enough, bnet_recv() realloc() the buffer. It will return an error (-2) if maxbytes is less than the record size sent.

It returns:

- >0: number of bytes read
- 0: on end of file
- -1: on hard end of file (i.e. network connection close)
- -2: on error

It should be noted that bnet_recv() is a blocking read.

15.9 bnet_sig

To send a "signal" from one daemon to another, one uses the subroutine:

int bnet_sig(BSOCK *sock, SIGNAL) where SIGNAL is one of the following:

- BNET_EOF - deprecated use BNET_EOD
- BNET_EOD - End of data stream, new data may follow
- BNET_EOD_POLL - End of data and poll all in one
- BNET_STATUS - Request full status
- BNET_TERMINATE - Conversation terminated, doing close()
- BNET_POLL - Poll request, I'm hanging on a read
- BNET_HEARTBEAT - Heartbeat Response requested
- BNET_HB_RESPONSE - Only response permitted to HB
- BNET_PROMPT - Prompt for UA

15.10 bnet_strerror

Returns a formated string corresponding to the last error that occurred.

15.11 bnet_close

The connection with the server remains open until closed by the subroutine:

void bnet_close(BSOCK *sock)

15.12 Becoming a Server

The bnet_open() and bnet_close() routines described above are used on the client side to establish a connection and terminate a connection with the server. To become a server (i.e. wait for a connection from a client), use the routine **bnet_thread_server**. The calling sequence is a bit complicated, please refer to the code in bnet_server.c and the code at the beginning of each daemon as examples of how to call it.

15.13 Higher Level Conventions

Within Bareos, we have established the convention that any time a single record is passed, it is sent with bnet_send() and read with bnet_recv(). Thus the normal exchange between the server (S) and the client (C) are:

```
S: wait for connection            C: attempt connection
S: accept connection              C: bnet_send() send request
S: bnet_recv() wait for request
S: act on request
S: bnet_send() send ack            C: bnet_recv() wait for ack
```

Thus a single command is sent, acted upon by the server, and then acknowledged.

In certain cases, such as the transfer of the data for a file, all the information or data cannot be sent in a single packet. In this case, the convention is that the client will send a command to the server, who knows that more than one packet will be returned. In this case, the server will enter a loop:

```
while ((n=bnet_recv(bsock)) > 0) {
   act on request
}
if (n < 0)
   error
```

The client will perform the following:

```
bnet_send(bsock);
bnet_send(bsock);
...
bnet_sig(bsock, BNET_EOD);
```

Thus the client will send multiple packets and signal to the server when all the packets have been sent by sending a zero length record.

CHAPTER

SIXTEEN

DIRECTOR CONSOLE OUTPUT

The console output should handle different *API modes*.

As the initial design of the Director Console connection did allow various forms of output (in fact: free text with some helper functions) the introduction of another API mode (API mode JSON) did require to change the creation of output.

- Output can be send to a console connection (UserAgent Socket) or the command is executed in a job
- free text (with the use of some helper functions)
- *Signals*
- Different Message types
  - Types
    - UAContext::send_msg
    - UAContext::info_msg
    - UAContext::warning_msg
    - UAContext::error_msg
  - indicated by a signal packet and than the text packet
- see *Daemon Protocols*

The `OUTPUT_FORMATTER` class have been introduced to consolidate the interface to generate Console output.

The basic idea is to provide an object, array and key-value based interface.

16.1 object_key_value

A main member function of `OUTPUT_FORMATTER` are the object_key_value(...) functions, like

```
void OUTPUT_FORMATTER::object_key_value(const char *key, const char *key_fmt, const␣
↪char *value, const char *value_fmt, int wrap = -1);
```

API mode 0 and 1 get the key and value, and write them as defined in the key_fmt and value_fmt strings. If the key_fmt string is not given, the key will not be written. If the value_fmt string is not given, the value will not be written.

In API mode 2 (JSON), OUTPUT_FORMATTER stores the key value pair in its internal JSON object, to delivers it, when the response object is finished. The keys will be transformed to lower case strings.

16.2 decoration

Additional output can be created by the `void OUTPUT_FORMATTER::decoration(const char *fmt,...)` function. This strings will only be written in API mode 0 and is intended to make an output prettier.

16.3 messages

For messages, the UAContext function should be used:

- UAContext::send_msg
- UAContext::info_msg
- UAContext::warning_msg
- UAContext::error_msg

Internally, these redirect to the `void OUTPUT_FORMATTER::message(const char *type,POOL_MEM &message)` function.

- API mode 0
  - packet 1: message is send to the user console
- API mode 1
  - packet 1: message signal is send
  - packet 2: message is send to the user console
- API mode 2:
  - packet 1: message signal is send (TODO: this might change)
  - message is stored in the corresponding message object
    - if an error message is send, the result of the command will switch to error

16.4 Objects and Arrays

To structure data, the `OUTPUT_FORMATTER` class offers functions:

- `void object_start(const char *name = NULL);`
- `void object_end(const char *name = NULL);`
- `void array_start(const char *name);`
- `void array_end(const char *name);`

These functions define the structure of the reult object in API mode 2, but are ignored in API mode 0 and 1.

16.4.1 named objects

- named objects (object_start(NAME))
  - can be added to objects (named and nameless objects)

16.4.2 nameless objects

- nameless objects (object_start(NULL))
  - can be added to arrays

16.4.3 arrays

- arrays (array_start(NAME))
  - can be added by name to an object
  - contain nameless objects (object_start(NULL))

16.5 Example

```
LockRes();
ua->send->array_start("storages");
foreach_res(store, R_STORAGE) {
    if (acl_access_ok(ua, Storage_ACL, store->name())) {
        ua->send->object_start();
        ua->send->object_key_value("name", store->name(), "%s\n");
        ua->send->object_end();
    }
}
ua->send->array_end("storages");
UnlockRes();
```

results to

```
*.api 2
{
  "jsonrpc": "2.0",
  "id": null,
  "result": {
    "api": 2
  }
}
*.storages
{
  "jsonrpc": "2.0",
  "id": null,
  "result": {
    "storages": [
      {
        "name": "File"
      },
      {
        "name": "myTapeLibrary"
      }
    ]
  }
}
```

16.6 Example with 3 level structure

```
ua->send->array_start("files");
for(int i=0; file[i]; i++) {
    ua->send->object_start();
    ua->send->object_key_value("Name", "%s=", file[i]->name, "%s");
    ua->send->object_key_value("Type", "%s=", file[i]->type, "%s");
    decode_stat(file[i]->lstat, &statp, sizeof(statp), LinkFI);
    ua->send->object_start("stat");
    ua->send->object_key_value("dev", "%s=", statp.st_dev, "%s");
    ua->send->object_key_value("ino", "%s=", statp.st_ino, "%s");
    ua->send->object_key_value("mode", "%s=", statp.st_mode, "%s");
    ...
    ua->send->object_end("stat");
    ua->send->object_end();
}
ua->send->array_end("files");
```

CHAPTER

SEVENTEEN

SMART MEMORY ALLOCATION

Few things are as embarrassing as a program that leaks, yet few errors are so easy to commit or as difficult to track down in a large, complicated program as failure to release allocated memory. SMARTALLOC replaces the standard C library memory allocation functions with versions which keep track of buffer allocations and releases and report all orphaned buffers at the end of program execution. By including this package in your program during development and testing, you can identify code that loses buffers right when it's added and most easily fixed, rather than as part of a crisis debugging push when the problem is identified much later in the testing cycle (or even worse, when the code is in the hands of a customer). When program testing is complete, simply recompiling with different flags removes SMARTALLOC from your program, permitting it to run without speed or storage penalties.

In addition to detecting orphaned buffers, SMARTALLOC also helps to find other common problems in management of dynamic storage including storing before the start or beyond the end of an allocated buffer, referencing data through a pointer to a previously released buffer, attempting to release a buffer twice or releasing storage not obtained from the allocator, and assuming the initial contents of storage allocated by functions that do not guarantee a known value. SMARTALLOC's checking does not usually add a large amount of overhead to a program (except for programs which use realloc() extensively; see below). SMARTALLOC focuses on proper storage management rather than internal consistency of the heap as checked by the malloc_debug facility available on some systems. SMARTALLOC does not conflict with malloc_debug and both may be used together, if you wish. SMARTALLOC makes no assumptions regarding the internal structure of the heap and thus should be compatible with any C language implementation of the standard memory allocation functions.

17.1 Installing SMARTALLOC

SMARTALLOC is provided as a Zipped archive, ; see the download instructions below.

To install SMARTALLOC in your program, simply add the statement:

to every C program file which calls any of the memory allocation functions (malloc, calloc, free, etc.). SMARTALLOC must be used for all memory allocation with a program, so include file for your entire program, if you have such a thing. Next, define the symbol SMARTALLOC in the compilation before the inclusion of smartall.h. I usually do this by having my Makefile add the "-DSMARTALLOC" option to the C compiler for non-production builds. You can define the symbol manually, if you prefer, by adding the statement:

#define SMARTALLOC

At the point where your program is all done and ready to relinquish control to the operating system, add the call:

sm_dump(*datadump*);

where *datadump* specifies whether the contents of orphaned buffers are to be dumped in addition printing to their size and place of allocation. The data are dumped only if *datadump* is nonzero, so most programs will normally use "sm_dump(0);". If a mysterious orphaned buffer appears that can't be identified from the information this prints about it, replace the statement with "sm_dump(1);". Usually the dump of the buffer's data will furnish the additional clues you need to excavate and extirpate the elusive error that left the buffer allocated.

Finally, add the files "smartall.h" and "smartall.c" from this release to your source directory, make dependencies, and linker input. You needn't make inclusion of smartall.c in your link optional; if compiled with SMARTALLOC not defined it generates no code, so you may always include it knowing it will waste no storage in production builds. Now when you run your program, if it leaves any buffers around when it's done, each will be reported by sm_dump() on stderr as follows:

```
Orphaned buffer:     120 bytes allocated at line 50 of gutshot.c
```

17.2 Squelching a SMARTALLOC

Usually, when you first install SMARTALLOC in an existing program you'll find it nattering about lots of orphaned buffers. Some of these turn out to be legitimate errors, but some are storage allocated during program initialisation that, while dynamically allocated, is logically static storage not intended to be released. Of course, you can get rid of the complaints about these buffers by adding code to release them, but by doing so you're adding unnecessary complexity and code size to your program just to silence the nattering of a SMARTALLOC, so an escape hatch is provided to eliminate the need to release these buffers.

Normally all storage allocated with the functions malloc(), calloc(), and realloc() is monitored by SMARTALLOC. If you make the function call:

```
sm_static(1);
```

you declare that subsequent storage allocated by malloc(), calloc(), and realloc() should not be considered orphaned if found to be allocated when sm_dump() is called. I use a call on "sm_static(1);" before I allocate things like program configuration tables so I don't have to add code to release them at end of program time. After allocating unmonitored data this way, be sure to add a call to:

```
sm_static(0);
```

to resume normal monitoring of buffer allocations. Buffers allocated while sm_static(1) is in effect are not checked for having been orphaned but all the other safeguards provided by SMARTALLOC remain in effect. You may release such buffers, if you like; but you don't have to.

17.3 Living with Libraries

Some library functions for which source code is unavailable may gratuitously allocate and return buffers that contain their results, or require you to pass them buffers which they subsequently release. If you have source code for the library, by far the best approach is to simply install SMARTALLOC in it, particularly since this kind of ill-structured dynamic storage management is the source of so many storage leaks. Without source code, however, there's no option but to provide a way to bypass SMARTALLOC for the buffers the library allocates and/or releases with the standard system functions.

For each function *xxx* redefined by SMARTALLOC, a corresponding routine named "actually*xxx*" is furnished which provides direct access to the underlying system function, as follows:

ll &
malloc(*size*) & actuallymalloc(*size*)
calloc(*nelem*, *elsize*) & actuallycalloc(*nelem*, *elsize*)
realloc(*ptr*, *size*) & actuallyrealloc(*ptr*, *size*)
free(*ptr*) & actuallyfree(*ptr*)

For example, suppose there exists a system library function named "getimage()" which reads a raster image file and returns the address of a buffer containing it. Since the library routine allocates the image directly with malloc(), you can't use SMARTALLOC's free(), as that call expects information placed in the buffer by SMARTALLOC's special version of malloc(), and hence would report an error. To release the buffer you should call actuallyfree(), as in this code fragment:

```
struct image *ibuf = getimage("ratpack.img");
display_on_screen(ibuf);
actuallyfree(ibuf);
```

Conversely, suppose we are to call a library function, "putimage()", which writes an image buffer into a file and then releases the buffer with free(). Since the system free() is being called, we can't pass a buffer allocated by SMARTALLOC's allocation routines, as it contains special information that the system free() doesn't expect to be there. The following code uses actuallymalloc() to obtain the buffer passed to such a routine.

```
struct image *obuf =
   (struct image *) actuallymalloc(sizeof(struct image));
dump_screen_to_image(obuf);
putimage("scrdump.img", obuf);  /* putimage() releases obuf */
```

It's unlikely you'll need any of the "actually" calls except under very odd circumstances (in four products and three years, I've only needed them once), but they're there for the rare occasions that demand them. Don't use them to subvert the error checking of SMARTALLOC; if you want to disable orphaned buffer detection, use the sm_static(1) mechanism described above. That way you don't forfeit all the other advantages of SMARTALLOC as you do when using actuallymalloc() and actuallyfree().

17.4 SMARTALLOC Details

When you include "smartall.h" and define SMARTALLOC, the following standard system library functions are redefined with the #define mechanism to call corresponding functions within smartall.c instead. (For details of the redefinitions, please refer to smartall.h.)

```
void *malloc(size_t size)
void *calloc(size_t nelem, size_t elsize)
void *realloc(void *ptr, size_t size)
void free(void *ptr)
void cfree(void *ptr)
```

cfree() is a historical artifact identical to free().

In addition to allocating storage in the same way as the standard library functions, the SMARTALLOC versions expand the buffers they allocate to include information that identifies where each buffer was allocated and to chain all allocated buffers together. When a buffer is released, it is removed from the allocated buffer chain. A call on sm_dump() is able, by scanning the chain of allocated buffers, to find all orphaned buffers. Buffers allocated while sm_static(1) is in effect are specially flagged so that, despite appearing on the allocated buffer chain, sm_dump() will not deem them orphans.

When a buffer is allocated by malloc() or expanded with realloc(), all bytes of newly allocated storage are set to the hexadecimal value 0x55 (alternating one and zero bits). Note that for realloc() this applies only to the bytes added at the end of buffer; the original contents of the buffer are not modified. Initializing allocated storage to a distinctive nonzero pattern is intended to catch code that erroneously assumes newly allocated buffers are cleared to zero; in fact their contents are random. The calloc() function, defined as returning a buffer cleared to zero, continues to zero its buffers under SMARTALLOC.

Buffers obtained with the SMARTALLOC functions contain a special sentinel byte at the end of the user data area. This byte is set to a special key value based upon the buffer's memory address. When the buffer is released, the key is

tested and if it has been overwritten an assertion in the free function will fail. This catches incorrect program code that stores beyond the storage allocated for the buffer. At free() time the queue links are also validated and an assertion failure will occur if the program has destroyed them by storing before the start of the allocated storage.

In addition, when a buffer is released with free(), its contents are immediately destroyed by overwriting them with the hexadecimal pattern 0xAA (alternating bits, the one's complement of the initial value pattern). This will usually trip up code that keeps a pointer to a buffer that's been freed and later attempts to reference data within the released buffer. Incredibly, this is *legal* in the standard Unix memory allocation package, which permits programs to free() buffers, then raise them from the grave with realloc(). Such program "logic" should be fixed, not accommodated, and SMARTALLOC brooks no such Lazarus buffer'' nonsense.

Some C libraries allow a zero size argument in calls to malloc(). Since this is far more likely to indicate a program error than a defensible programming stratagem, SMARTALLOC disallows it with an assertion.

When the standard library realloc() function is called to expand a buffer, it attempts to expand the buffer in place if possible, moving it only if necessary. Because SMARTALLOC must place its own private storage in the buffer and also to aid in error detection, its version of realloc() always moves and copies the buffer except in the trivial case where the size of the buffer is not being changed. By forcing the buffer to move on every call and destroying the contents of the old buffer when it is released, SMARTALLOC traps programs which keep pointers into a buffer across a call on realloc() which may move it. This strategy may prove very costly to programs which make extensive use of realloc(). If this proves to be a problem, such programs may wish to use actuallymalloc(), actuallyrealloc(), and actuallyfree() for such frequently-adjusted buffers, trading error detection for performance. Although not specified in the System V Interface Definition, many C library implementations of realloc() permit an old buffer argument of NULL, causing realloc() to allocate a new buffer. The SMARTALLOC version permits this.

17.5 When SMARTALLOC is Disabled

When SMARTALLOC is disabled by compiling a program with the symbol SMARTALLOC not defined, calls on the functions otherwise redefined by SMARTALLOC go directly to the system functions. In addition, compile-time definitions translate calls on the "actually...()" functions into the corresponding library calls; "actuallymalloc(100)", for example, compiles into"malloc(100)''. The two special SMARTALLOC functions, sm_dump() and sm_static(), are defined to generate no code (hence the null statement). Finally, if SMARTALLOC is not defined, compilation of the file smartall.c generates no code or data at all, effectively removing it from the program even if named in the link instructions.

Thus, except for unusual circumstances, a program that works with SMARTALLOC defined for testing should require no changes when built without it for production release.

17.6 The alloc() Function

Many programs I've worked on use very few direct calls to malloc(), using the identically declared alloc() function instead. Alloc detects out-of-memory conditions and aborts, removing the need for error checking on every call of malloc() (and the temptation to skip checking for out-of-memory).

As a convenience, SMARTALLOC supplies a compatible version of alloc() in the file alloc.c, with its definition in the file alloc.h. This version of alloc() is sensitive to the definition of SMARTALLOC and cooperates with SMARTALLOC's orphaned buffer detection. In addition, when SMARTALLOC is defined and alloc() detects an out of memory condition, it takes advantage of the SMARTALLOC diagnostic information to identify the file and line number of the call on alloc() that failed.

17.7 Overlays and Underhandedness

String constants in the C language are considered to be static arrays of characters accessed through a pointer constant. The arrays are potentially writable even though their pointer is a constant. SMARTALLOC uses the compile-time definition ./smartall.wml to obtain the name of the file in which a call on buffer allocation was performed. Rather than reserve space in a buffer to save this information, SMARTALLOC simply stores the pointer to the compiled-in text of the file name. This works fine as long as the program does not overlay its data among modules. If data are overlayed, the area of memory which contained the file name at the time it was saved in the buffer may contain something else entirely when sm_dump() gets around to using the pointer to edit the file name which allocated the buffer.

If you want to use SMARTALLOC in a program with overlayed data, you'll have to modify smartall.c to either copy the file name to a fixed-length field added to the abufhead structure, or else allocate storage with malloc(), copy the file name there, and set the abfname pointer to that buffer, then remember to release the buffer in sm_free. Either of these approaches are wasteful of storage and time, and should be considered only if there is no alternative. Since most initial debugging is done in non-overlayed environments, the restrictions on SMARTALLOC with data overlaying may never prove a problem. Note that conventional overlaying of code, by far the most common form of overlaying, poses no problems for SMARTALLOC; you need only be concerned if you're using exotic tools for data overlaying on MS-DOS or other address-space-challenged systems.

Since a C language "constant'' string can actually be written into, most C compilers generate a unique copy of each string used in a module, even if the same constant string appears many times. In modules that contain many calls on allocation functions, this results in substantial wasted storage for the strings that identify the file name. If your compiler permits optimization of multiple occurrences of constant strings, enabling this mode will eliminate the overhead for these strings. Of course, it's up to you to make sure choosing this compiler mode won't wreak havoc on some other part of your program.

17.8 Test and Demonstration Program

A test and demonstration program, smtest.c, is supplied with SMARTALLOC. You can build this program with the Makefile included. Please refer to the comments in smtest.c and the Makefile for information on this program. If you're attempting to use SMARTALLOC on a new machine or with a new compiler or operating system, it's a wise first step to check it out with smtest first.

17.9 Invitation to the Hack

SMARTALLOC is not intended to be a panacea for storage management problems, nor is it universally applicable or effective; it's another weapon in the arsenal of the defensive professional programmer attempting to create reliable products. It represents the current state of evolution of expedient debug code which has been used in several commercial software products which have, collectively, sold more than third of a million copies in the retail market, and can be expected to continue to develop through time as it is applied to ever more demanding projects.

The version of SMARTALLOC here has been tested on a Sun SPARCStation, Silicon Graphics Indigo2, and on MS-DOS using both Borland and Microsoft C. Moving from compiler to compiler requires the usual small changes to resolve disputes about prototyping of functions, whether the type returned by buffer allocation is char or void , and so forth, but following those changes it works in a variety of environments. I hope you'll find SMARTALLOC as useful for your projects as I've found it in mine.

CHAPTER

EIGHTEEN

GNU FREE DOCUMENTATION LICENSE

Version 1.2, November 2002

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

CHAPTER

NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search